ARM Software Development Toolkit Version 2.11

Reference Guide

Document Number: ARM DUI 0041B
Issued: June 1997

ENGLAND
Advanced RISC Machines Limited
Fulbourn Road
Cherry Hinton
Cambridge CB1 4JN
England
Telephone:+44 1223 400400
Facsimile:+44 1223 400410
Email:info@armltd.co.uk

GERMANY
Advanced RISC Machines Limited
Otto-Hahn Str. 13b
85521 Ottobrunn-Riemerling
Munich
Germany
Telephone:+49 (0) 89 608 75545
Facsimile:+49 (0) 89 608 75599
Email:info@armltd.co.uk

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

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

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

Proprietary Notice

Change Log

Issue	Date	By	Change
A	Jan 97	BJH	Created from ARM DUI 0020; includes major updates for SDT 210.
B	June 97	BJH	Updated for SDT 211.

Preface

This preface introduces the ARM Software Development Toolkit and its documentation.

Preface

About This Manual

Overview

This manual covers the following topics:

- the components of the ARM Software Development Toolkit
- reference information on each of the ARM Software Tools
- procedure call standards
- file formats

Note *This manual does not cover device-specific issues. Please refer to the appropriate ARM device datasheet.*

Organization

This *Reference Guide* is organized into the following parts:

Part 1 **Toolkit Reference**
Gives reference information on the individual tools in the toolkit.
For example: file-naming conventions, command-line options, variables, procedure call standards, and utilities.

Part 2 **Debug Reference**
Gives reference information on ARM's debugging tools.
For example: Angel, ARMulators, and ARM Symbolic Debugger.

Part 3 **File Format Reference**
Gives information on the file formats in use in the toolkit.
For example: ARM Object Format, ARM Image Format, ARM Object Library Format, ELF.

Typographical Conventions

Typographical conventions

The following typographical conventions are used in this manual:

`typewriter`	Denotes text that may be entered at the keyboard: commands, file and program names, and assembler and C source.
`typewriter`	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`	Shows text which must be substituted with user-supplied information. This is most often used in syntax descriptions.
Oblique	Highlights important notes and ARM-specific terminology.

Thumb Boxes like this contain information that applies specifically to Thumb-aware variants of the ARM Software Development Toolkit.

Filenames

Unless otherwise stated, filenames are quoted in MS-DOS format, for example:

```
EXAMPLES\BASICASM\GCD1.S
```

If you are using the UNIX platform, you must translate them into their UNIX equivalent:

```
examples/basicasm/gcd1.s
```

Release Components

Programming and modeling tools

The following tools are described in full in the relevant chapters of this manual. Please note that your release of the Toolkit may not include all the tools mentioned below; see the Release Notes for a definitive list of the tools supplied with your release.

`armcc`	The ARM C compiler	See ***Chapter 1, C Compilers***
`tcc`	The Thumb C compiler	See ***Chapter 1, C Compilers***
`armasm`	The ARM assembler	See ***Chapter 2, Assembler***
`tasm`	The Thumb assembler	See ***Chapter 2, Assembler***
`armlink`	The ARM linker	See ***Chapter 3, Linker***
`decaof`	The ARM–Thumb object-file decoder/ disassembler	See ***Chapter 7, Toolkit Utilities***
`decaxf`	The ARM Executable format decoder.	See ***Chapter 7, Toolkit Utilities***
`armlib`	The ARM object-file librarian	See ***Chapter 7, Toolkit Utilities***
`armsd`	The ARM command-line debugger	See ***Chapter 10, ARM Debugger***

Retargetable libraries

Three retargetable libraries are supplied:

- The ARM ANSI C library, supplied in both source form and as an object library.
- The ARM embedded C library, supplied in both source form and as an object library.

For further information see ***Chapter 4, Rebuilding the C Library***.

Thumb The Thumb 16-bit ANSI C library is provided as an object library in both little-endian and big-endian forms.

Feedback

Feedback on the ARM Software Development Toolkit

If you have comments or suggestions about the ARM Software Development Toolkit, please contact your supplier, giving:

- details of which platform and release of the ARM software tools you are using
- a small sample code fragment which illustrates your comment
- precise description of your comment or suggestion

Feedback on this manual

If you have feedback on this manual, please contact your supplier, giving:

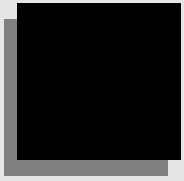
- the manual's title
- the manual's document number
- the page number(s) to which your comments refer
- a concise explanation of the comment

General suggestions for additions and improvements are also welcome.

Contents

Contents

Contents



Tools Reference

1 C Compilers

This chapter describes the ARM and Thumb C compilers.

1.1 Introduction

This chapter includes the reference information you need to make effective use of the ARM C system. For an introduction to compiling and linking an ARM C program, see the *Software Development Toolkit User Guide (ARM DUI 0040)*.

Note *This chapter is not intended to be an introduction to C and does not try to teach programming in C, nor is it a reference manual for the C standard.*

1.1.1 Recommended texts

C programming guides

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

- Kernighan, B.W. and Ritchie, D.M., *The C Programming Language* (2nd edition, 1988). Prentice-Hall, Englewood Cliffs, NJ, USA. ISBN 0-13-110362-8.
  This is the original C bible, updated to cover the essentials of ANSI C.
- Harbison, S.P. and Steele, G.L., *A C Reference Manual* (second edition, 1987). Prentice-Hall, Englewood Cliffs, NJ, USA. ISBN 0-13-109802-0.
  This is a very thorough reference guide to C, including a useful amount of information on ANSI C.
- Koenig, A, *C Traps and Pitfalls*, Addison-Wesley (1989), Reading, Mass. ISBN 0-201-17928-8.
  This explains how to avoid the most common traps and pitfalls in C programming. It provides informative reading at all levels of competence in C.

ANSI C reference

- ISO/IEC 9899:1990, *C Standard*.
  This is available from ANSI as X3J11/90-013. The standard is available from the national standards body (eg. AFNOR in France, ANSI in the USA).

1.2 About the ARM C Compilers

The ARM C compiler compiles both ANSI C and the dialect of C used by Berkeley UNIX.

Wherever possible, it adopts widely-used command-line options which are familiar to users of both UNIX and DOS.

1.2.1 Compiler variants

There are two variants of the ARM C compiler:

armcc	compiles C source into 32-bit ARM code
tcc	compiles C source into 16-bit Thumb code

As they have the same basic front end, the descriptions in this chapter apply to both. Where tcc has added features or restrictions, these are dealt with in Thumb-specific sections.

Note *If you want to link compiled ARM and Thumb code together, refer to the Software Development Toolkit User Guide (ARM DUI 0040).*

1.2.2 Source language modes

By default, the ARM C Compiler compiles ANSI C as defined by ISO/IEC 9899:1990–C Standard

ANSI mode

In ANSI mode, the ARM C compiler has been tested against release 7.00 of the Plum-Hall C Validation Suite (CVS) which has been adopted by the British Standards Institute for C compiler validation in Europe. In the language conformance sections of the CVS, it differs in only two trivial ways; both relate to producing the required diagnostics:

- an empty initializer for an aggregate of complete type is not diagnosed. For example:

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

- a signed integer constant overflow is not treated as an error, but merely warned of. For example:

  ```
  case INT_MAX+1: ...
  ```

The ANSI command-line option is described on page 1-9.

pcc mode

Select pcc mode from the compiler's command line to accept the dialect of C used by Berkeley UNIX. This is described on page 1-9.

1.3 File Usage

This section introduces some key concepts.

1.3.1 Naming conventions

The ARM C system uses suffix-naming conventions to identify the classes of file involved in the compilation and linking processes:

`.c`	C source file
`.h`	C header file
`.o`	ARM object file
`.s`	ARM or Thumb assembly language
`.lst`	compiler listing file

For example, `testfile.c` names the C source of the file called `testfile`.

Many host systems support suffix-naming conventions (UNIX, MS-DOS), so the names used by the C system on the command line, and as arguments to the C preprocessor `#include` directive, map directly to host filenames.

Portability

The ARM C system supports the use of multiple filenaming conventions on one host. In general, follow these guidelines:

- restrict the name of a file or directory to a maximum of eight lowercase letters and digits, beginning with a letter
- ensure that extensions are no more than three letters and digits in length
- make embedded pathnames relative, rather than absolute

In each environment, the ARM C system supports:

- native filenames
- pseudo UNIX filenames, which have the format:

  `host-volume-name``:/`*`rest-of-unix-file-name`*
- UNIX filenames

Names are parsed as follows:

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

This filename interpretation succeeds only if you adhere to certain rules, such as filename length.

1.3.2 Specifying keyboard input

Specifying `-` (minus) as the source filename takes input from the keyboard. It is terminated by entering Ctrl D.

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

1.3.3 Filename validity

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

1.3.4 Object files

By default, the object file(s) created by the compiler are stored in the current directory.

A C source file (`file.c`) is compiled into an object file (`file.o`) written in ARM Object Format (AOF). AOF is defined in ***14.1 ARM Object Format*** on page 14-2.

1.3.5 Included files

During a compilation, the compiler may read included header files, conventionally given a `.h` suffix, or included C source files, usually given a `.c` suffix.

A special feature of the ARM C system is that the ANSI library headers are built into the C compiler (in a special, textually-compressed, in-memory filing system) and are used from there by default. Placing a filename in angle brackets indicates that the included file is a system file and ensures that the compiler looks first in its built-in filing system:

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

Enclosing a filename in double quotes in the `#include` directive indicates that it is not a system file. In this example, the ARM C compiler looks for the specified file in the current directory, by default:

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

The way the compiler looks for included files depends on three factors:

- whether the filename is an absolute filename, rather than a relative filename
- whether the filename is between angle brackets or double quotes
- use of the `-I` and `-j` flags and the special directory name `:mem`

The current place

The *current place* is the directory containing the source file (C source or `#include` header) currently being processed by the compiler. This is often the current directory.

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, suppose that the current place is `\me\include` and the compiler is seeking the `#include` file `sys\defs.h`. This is found as:

```
\me\include\sys\defs.h
```

The new current place is now `\me\include\sys` and any file `#included` by `defs.h`, whose name is not absolute, is sought relative to `\me\include\sys`.

This is the search rule used by Berkeley UNIX systems.

If required, the stacking of current places can be disabled with the compiler option `-fK`, which makes the compiler use the search rule described originally 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 being compiled.

The search path

The order of directories on the search path is:

1. the compiler's own in-memory filing system (for filenames enclosed in angle brackets, but only if the `-j` flag is not used)
2. the current place (see above) (not for filenames enclosed in angle brackets)
3. arguments to `-I` flags, if used (for filenames enclosed in angle brackets or double quotes)
4. arguments to the `-j` flag, if used (for filenames enclosed in angle brackets or double quotes)
5. the compiler's own in-memory filing system (for filenames enclosed in angle brackets, but only if the `-j` flag is used)

1.4 Command Syntax

The general form of the command for invoking the C compiler is one of:

```
armcc options sourcefile
tcc options sourcefile
```

By default, the C compiler looks for source files, and creates object, assembler, and listing files in the current directory.

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

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

The command-line options are divided into the following subsections, so that options controlling related aspects of the compiler's operation are grouped together:

- using the ARM Procedure Call Standard (APCS)
- controlling the linker
- selecting processors
- using preprocessor flags
- controlling code generation
- controlling warning messages
- suppressing error messages
- controlling additional compiler features

Getting help

The `-help` option gives a summary of the compiler's command-line options.

1.4.1 APCS command-line options

You use the following command-line option to specify which variant of the ARM Procedure Call Standard (APCS) is to be used by the compiler.

`-apcs [3]`*`quals`*

Notes

- There must be a space between `-apcs` and the first qualifier.
- At least one qualifier must be present, and there must be no space between qualifiers. The qualifiers are listed on the next page.

APCS qualifiers

Thumb The options marked with a * are not applicable to Thumb.

APCS variants

`/26bit`	*	26-bit APCS variant.
`/32bit`	*	32-bit APCS variant.
`/reentrant`	*	Re-entrant APCS variant.
`/nonreentrant`	*	Non re-entrant APCS variant.

Stack checking

`/swstackcheck`	Software stack-checking APCS variant. This is the default for ARM.
`/noswstackcheck`	No software stack-checking APCS variant. This is the default for Thumb.

Frame pointers

`/fp`	*	Use a dedicated frame-pointer register.
`/nofp`	*	Do not use a frame-pointer.

Floating-point compatibility

`/fpe2`	*	Floating-point emulator 2 compatibility.
`/fpe3`	*	Floating-point emulator 3 compatibility.
`/fpregargs`	*	Floating-point arguments passed in floating-point registers.
`/nofpregargs`	*	Floating-point arguments are not passed in floating-point registers.
`/softfp`	*	Call software floating-point library functions. **Note:** *This is the default for ARM and the only floating-point method available for Thumb.*
`/hardfp`	*	Generate ARM coprocessor instructions for floating-point (may also specify `fpe2`/`fpe3` and `fpr`/`nofpr`).

ARM/Thumb interworking

`/interwork`	Compile code for ARM/Thumb interworking. See the *Software Development Toolkit User Guide*.
`/nointerwork`	Do not compile code which is suitable for ARM/Thumb interworking. This is the default.

Narrow parameters

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

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

1.4.2 Setting endianness

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

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

1.4.3 Setting the source language

`-ansi` Compiles ANSI standard C (on by default).

`-fc` Enables the “limited pcc” option, designed to support the use of pcc-style header files in an otherwise strict ANSI mode (for example, when using libraries of functions implemented in old-style C from an application written in ANSI C). This allows characters after `#else` and `#endif` preprocessor directives (which are ignored).
The “limited pcc” option also supports system programming in ANSI mode by suppressing warnings about explicit casts of integers to function pointers, and permitting the dollar character in identifiers; linker-generated symbols often contain “$$”, and all external symbols containing “$$” are reserved to the linker.

`-fussy` Is extra strict about enforcing conformance to ANSI standard or pcc conventions (for example, prohibit the `volatile` qualifier in pcc mode).

`-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 on UNIX systems. The `-pcc` keyword alters the language accepted by the compiler, but the built-in ANSI headers are still used.

`-pedantic` See `-fussy`.

`-strict` See `-fussy`.

1.4.4 Working with files

`-errors` *`file`*	Writes the compiler error output to *`file`*.
`-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 using the `-f` flag (see ***1.4.15 Miscellaneous compiler features*** on page 1-18).
`-via` *`file`*	Opens the specified file and reads in more command-line arguments from it.

1.4.5 Search paths

`-I`*`dir-name`*	Adds the specified directory to the list of places which are searched for included files (*after* the in-memory or source file directory). The directories are searched in the order they are given, by multiple `-I` options. See ***1.3.5 Included files*** on page 1-5 for full details. The in-memory filing system is specified by `:mem`.
`-fk`	Uses Kernighan and &Ritchie search rules for locating included files (the current place is defined by the original source file and is not stacked; see ***The current place*** on page 1-5 for details). If you do not use this option, Berkeley-style searching is used.
`-fd`	Makes the handling of "..." included files the same as <...> included 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 (for example, after all directories specified by `-I` options), but otherwise works in the same way as `-I`. It also forces the compiler to search the in-memory filing system *after* all other searches have failed. The in-memory filing system is specified by `:mem`. `-j` is an ARM-specific flag and is not portable to other C systems. You cannot have more than one `-j` option on a command line. See ***1.3.5 Included files*** on page 1-5 for full details.

1.4.6 Controlling the linker

`-c`
: Does not perform the link step. This just compiles the source program(s), leaving the object file(s) in the current directory (or as directed by the `-o` flag). Note that this option is different from the `-C` (uppercase) option, which is described on page 1-14.

`-fe`
: Checks that external names used within the file are still unique when reduced to six case-insensitive characters. Some linkers support as few as six significant characters in external symbol names. This can cause problems with clashes if a system uses two names such as `getExpr1` and `getExpr2`, which are only unique in the eighth character. The check can only be made within a single compilation unit (source file), so it cannot catch all such problems. Since ARM C allows external names of up to 256 characters, this is strictly an aid to portability.

1.4.7 Controlling code generation

`-o` *`file`*
: Names the file which holds the final output of the compilation step.
  - In conjunction with `-c`, it gives the name of the object file.
  - In conjunction with `-S`, it gives the name of the assembly language file.
  - Otherwise, it names the final output of the link step.

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

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

`-S`
: Does not generate object code, but the compiler writes a listing of the assembly language to a file.The filename defaults to *`file`*`.s` in the current directory (where *`file`*`.c` is the name of the source file stripped of any leading directory names). The default can be overridden using the `-o` flag.

1.4.8 Debug information

`-asd` Uses asd table format (default).

`-dwarf` Uses dwarf table format.

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

- `-gt` All available entries should be generated.
- `-gtp` Tables should not include pre-processor macrodefinitions. (Ignored if DWARF debug tables are generated, as there is then no way to describe macros.

`-gx`*`letters`* Specifies the level of optimisation allowed when generating debug tables.

- `-gx` No optimisations
- `-gxr` Unoptimised register allocation
- `-gxo` Full optimisation
- Note 1 You must take care with the values of local variables, as many local variables may occupy the same register and you might get unexpected values displayed in the debugger.
- Note 2 The code generated with `-gxo` in force is still somewhat degraded compared with that generated when no debug tables are generated, as some optimisations normally performed by the compiler cannot be described within the debug table formats used, and so are always disabled.

`-g+` Switches the generation of debug tables on for the current compilation (with options as specified by `-gt` and `-gx`).

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

Note *The effect of* `-gt<T-`*`options`*`>`*,* `-gx<X-`*`options`*`>` *and* `-g+` *may be combined in the single command-line argument:*

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

1.4.9 Processor selection

`-processor` *name*
: Compiles code for the specified processor. The compiler may take advantage of certain features of the selected processor which may make the code incompatible with other processors; for example, the use of halfword instructions.

  name is the processor number; for example:

  `ARM2`
  `ARM3`
  `ARM6` (default)
  `ARM7`
  `ARM7M`
  `ARM7TM`
  `ARM8`
  `StrongARM1`

`-architecture` *n*
: Specifies the ARM architecture version that compiled code will comply with. The options are:

  - `3`
  - `3M`
  - `4`
  - `4T`

Halfword support

To enable halfword support, use option `-architecture 4` or `-architecture 4T`, or specify an appropriate processor using `-processor`.

Thumb

Specifying a Thumb-aware processor (eg. `-processor ARM7TM`) to armcc does not make `armcc` generate Thumb code. Instead, it generates ARM code which uses the Architecture 4 halfword load and store ARM instructions. This option is not available with tcc, as tcc always generates Thumb code.

1.4.10 Preprocessor flags

Note *In the following list, * means that the option can be repeated many times.*

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

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

where *`toolname`* is either `armcc` or `tcc`. By default, comments are stripped from the output (but see the `-C` flag, below).

`-C`
Retains comments in preprocessor output when used in conjunction with `-E`. Note that this option is different from the `-c` (lowercase) flag, which suppresses the link step. See page 1-11 for a description of the `-c` option.

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

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

where *`toolname`* is either `armcc` or `tcc`.

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

```
#define symbol value
```

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

```
#define symbol
```

The symbol is given the default value 1.

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

```
#undef symbol
```

1.4.11 Controlling warning messages

This section describes how you turn warning messages on or off.

`-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.

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

The `-W` option controls the suppression of warning messages. The compiler uses warnings to indicate potential portability problems or other hazards. You can avoid having too many warning messages in the early stages of porting a program written in old-style C by disabling warnings. The options are on by default, unless specified otherwise.

Note *If the + character is included in the characters following the W flag, the warnings corresponding to any following letters are enabled rather than supresssed.*

`-W` Suppresses all warnings. If one or more letters follow the flag, only the warnings controlled by those letters are suppressed.
The following example suppresses the warnings controlled by `a` and `d`, and enables those controlled by `f` and `g`:

```
-Wad+fg
```

`-Wa` Suppresses the following warning:

```
Use of = in a condition context
```

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

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

where it is possible that the author really intended:

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

or that the author intended the following, but missed a keystroke:

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

In new code, avoid the deliberate use of `if (a = b) ...`
This warning is suppressed by default in pcc mode.

`-Wd` Suppresses the following message, given when a declaration without argument types is encountered in ANSI mode (the warning is suppressed by default in pcc mode).

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

In ANSI C, declarations like this are deprecated, and a future version of the C standard may ban them. They are already illegal in C++. However, it is sometimes useful to suppress this warning when porting old code.

`-Wf` Suppresses the following message:

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

which may be useful when compiling old-style C in ANSI mode. This is suppressed by default in pcc mode.

`-Wg` Header file not guarded. This is off by default.

`-Wl` Lower precision in wider context. This option is off by default. This warning arises in cases like:

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

where the multiplication yields an int result which is then widened to long. Because int and long have the same length for ARM and THUMB, this only warns of portability problems to targets with 16-bit ints or 64-bit longs.

`-Wn` Suppresses the following warning:

```
Implicit narrowing cast
```

This warning is issued when the compiler detects the implicit narrowing of a long expression in an `int` or `char` context, or the implicit narrowing of a floating-point expression in an integer or narrower floating-point context. Such implicit narrowings are almost always a source of problems when moving code developed using a fully 32-bit system (such as ARM C) to a system in which ints occupy 16 bits and longs 32 bits. This is suppressed by default in PCC mode.

`-Wp` Suppresses the following warning:

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

ANSI requires that you only use `#include <...>` for ANSI headers, but 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 `-strict` is in use.

`-Ws` Padding inserted in struct. This is off by default.

`-Wu` Suppresses warnings about future compatibility (ie. C++) for both armcc and tcc. This option is off by default.

`-Wv` Suppresses the following warning:

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

This is most often caused by a return from a function which was assumed to return int (because no other type was specified) but is being used as a void function. As this is widespread in old-style C, it is suppressed by default in pcc mode.

1.4.12 Suppressing error messages

These options force the compiler to accept C source which would normally produce errors. If you use any of these options, it means that the C source does not conform to the ANSI C standard (the compiler normally generates precisely the diagnostics required by ANSI).

These options are on by default unless specified otherwise, and + is interpreted as for `-W`.

`-ec` Suppresses all implicit cast errors. For example;

```
implicit cast of nonzero int to pointer
```

`-ef` Suppresses errors for unclean casts such as short to pointer.

`-el` Suppresses errors about linkage disagreements where functions are implicitly declared extern and later defined as static.

`-ep` Suppresses the error which occurs if there are extraneous characters at the end of a preprocessor line. This option is suppressed in PCC mode.

`-ez` Suppresses the error which occurs if a zero-length array is used.

1.4.13 Load and store options

`-za`*`Option`* Specifies whether LDR may only access word-aligned addresses:

`1`	Yes
`0`	No

`-zr`*`Number`* Allows the size of most load multiple and all store instructions to be controlled between the limits of 3 and 16 registers transferred. This can help control interrupt latency where this is critical. *`Number`* defaults to 16.
Note: The Thumb compiler (tcc) does not currently support this option.

1.4.14 Alignment options

`-zap`*`Number`* Specifies whether pointers to structs are assumed to be aligned on at least <struct minimal alignment> boundaries:

`1`	Yes
`0`	No. Casting `short[ ]` to struct `(short, short,...)` does not cause a problem.

`-zas`*`Number`* Specifies the minimal byte alignment for structs. Permitted values for *`Number`* are as follows (default is 4):

```
1, 2, 4, 8
```

`-zat`*`Number`* Specifies the minimal byte alignment for top-level static objects, such as global variables. Permitted values for *`Number`* are as follows (default 4):

```
1, 2, 4, 8
```

1.4.15 Miscellaneous compiler features

There are a number of additional compiler features which control areas such as code generation and special portability options.

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

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

`-zo` Generates one AOF area per function.

`-zp`*`LetterDigit`* Emulates `#pragma` directives. The letter and digit which follow it are the same characters that would follow the "-" of a `#pragma` directive. See ***1.9.1 Pragmas*** on page 1-46 for details.

The `-f` option described below controls a variety of compiler features, including certain checks more rigorous than usual. It is followed by a string of modifier letters. At least one letter is required, though several may be given at once. For example:

```
-ffh
```

Note *If the + character is included in the modifier letters, the effect of any following letters is disabled, rather than enabled.*

When writing high-quality production software, you are encouraged to use at least the `-fh` options in the later stages of program development (the extra diagnostics produced can be annoying in the earlier stages).

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

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

`-fi` Lists (in the listing file) the lines from any files included with directives of the form:

```
#include "file"
```

`-fj` As for `-fi`, but for files included by lines of the form:

```
#include <file>
```

`-fn` Embeds function names in the code area (see `-ff` option). This improves the readability of the output produced by the stack backtrace runtime support function and the `_mapstore()` function. However, it does increase the size of the code area by around 5%. In general, it is not useful to specify `-ff` with `-p` (see ***1.4.7 Controlling code generation*** on page 1-11). This option is disabled by default.

`-fp` Reports on explicit casts of integers into 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 where both are 32-bit types. (Implicit casts are reported anyway, unless suppressed by the `-Wc` option.)

`-fu` 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 user's unexpanded source text is listed. For example, consider the line:

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

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

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

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

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

1.5 Implementation Details

This section gives details of those aspects of the compiler and C library which the ANSI standard for C identifies as implementation-defined, together with other points of interest.

1.5.1 Data Elements

Integers are represented in two's complement form.

Data items of type char are unsigned by default, though in ANSI mode they may be explicitly declared as signed char or unsigned char.

In the compiler's pcc mode there is no `signed` keyword, so chars are signed by default and may be declared unsigned if required.

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

Type	Size in bits
char	8
short	16
int	32
long	32
float	32
double	64
long double	64 (subject to change)
all pointers	32

Table 1-1: Size of data elements

1.5.2 Character sets and identifiers

An identifier can be of any length. The compiler truncates an identifier after 256 characters, all of which are significant (the standard requires a minimum of 31 significant characters).

The source character set expected by the compiler is 7-bit ASCII. Within comments, string literals, and character constants, the full ISO 8859-1 (Latin 1) 8-bit character set is recognised.

In its generic configuration, as delivered, the C library processes the full ISO 8859-1 (Latin-1) 8-bit character set, except that the default locale is the C locale (see ***1.6 Standard Implementation Definition*** on page 1-27). The `ctype` functions therefore all return 0 when applied to codes in the range 160 to 255.

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.

Uppercase and lowercase characters are distinct in all internal and external identifiers.

In pcc mode (`-pcc` option) and "limited pcc" or "system programming" mode (`-fc` option), an identifier may also contain a dollar character.

1.5.3 Arithmetic limits (limits.h and float.h)

The ANSI C standard defines two header files (`limits.h` and `float.h`) which contain constants describing the ranges of values that can be represented by the arithmetic types. The standard also defines minimum values for many of these constants. This subsection gives the values and significance of these two headers for the ARM.

Number of bits in the smallest object that is not a bit field (ie. a byte):

```
CHAR_BIT 8
```

Maximum number of bytes in a multibyte character, for any supported locale:

```
MB_LEN_MAX 1
```

For the following integer ranges, the middle column gives the numerical value of the range's endpoint, while the right hand column gives the bit pattern (in hexadecimal) that would be interpreted as this value in ARM C. When entering constants, you must be careful about the size and signed-ness of the quantity. Constants are interpreted differently in decimal and hexadecimal/octal. See the ANSI C standard or any of the recommended textbooks on the C programming language for more details.

Range	End-point	Hex Representation
CHAR_MAX	255	0xff
CHAR_MIN	0	0x00
SCHAR_MAX	127	0x7f
SCHAR_MIN	-128	0x80
UCHAR_MAX	255	0xff
SHRT_MAX	32767	0x7fff
SHRT_MIN	-32768	0x8000
USHRT_MAX	65535	0xffff
INT_MAX	2147483647	0x7fffffff
INT_MIN	-2147483648	0x80000000
LONG_MAX	2147483647	0x7fffffff
LONG_MIN	-2147483648	0x80000000
ULONG_MAX	4294967295	0xffffffff

Table 1-2: ARM C compiler integers ranges

Characteristics of floating point

```
FLT_RADIX       2
FLT_ROUNDS      .1
```

The base (radix) of the ARM floating-point number representation is 2, and floating-point addition rounds to nearest.

Ranges of floating types

```
FLT_MAX   3.40282347e+38F
FLT_MIN   1.17549435e-38F
DBL_MAX   1.79769313486231571e+308
DBL_MIN   2.22507385850720138e-308
LDBL_MAX  1.79769313486231571e+308
LDBL_MIN  2.22507385850720138e-308
```

Ranges of base two exponents

```
FLT_MAX_EXP          128
FLT_MIN_EXP        (-125)
DBL_MAX_EXP         1024
DBL_MIN_EXP       (-1021)
LDBL_MAX_EXP        1024
LDBL_MIN_EXP      (-1021)
```

Ranges of base ten exponents

```
FLT_MAX_10_EXP        38
FLT_MIN_10_EXP      (-37)
DBL_MAX_10_EXP        308
DBL_MIN_10_EXP     (-307)
LDBL_MAX_10_EXP       308
LDBL_MIN_10_EXP    (-307)
```

Decimal digits of precision

```
FLT_DIG                6
DBL_DIG               15
LDBL_DIG              15
```

Digits (base two) in mantissa (binary digits of precision)

```
FLT_MANT_DIG          24
DBL_MANT_DIG          53
LDBL_MANT_DIG         53
```

Smallest positive values such that (1.0 + x != 1.0)

```
FLT_EPSILON            1.19209290e-7F
DBL_EPSILON            2.2204460492503131e-16
LDBL_EPSILON           2.2204460492503131e-16L
```

1.5.4 Structured data types

The ANSI C standard leaves details of the layout of the components of a structured data type to each implementation. The following points apply to the ARM C compiler:

- The alignment of a structure is the larger of:
  - the maximum alignment required by any of its members
  - the minimal alignment for all structs (as defined by `-zat`, which is described in ***1.4.14 Alignment options*** on page 1-17).
- Structures are arranged with the first-named component at the lowest address.
- A component with a char type is packed into the next available byte.
- A component with a short type is aligned to the next even-addressed byte.
- All other arithmetic-type components are word-aligned, as are pointers and integers containing bitfields.
- Except for `-strict` (when the only valid types for bitfields are signed int and unsigned int), bitfields may have any integral type.
- A bitfield whose type includes neither the signed or unsigned qualifier, is treated as having the same signedness as plain char, unsigned by default (signed by default in `-pcc` mode).
- A bitfield must be wholly contained within a correctly aligned object of tis type.
- Bitfields are allocated within words 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

1.5.5 Pointers

The following remarks apply to pointer types:

- Adjacent bytes have addresses which 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 (but not in `-pcc` mode).

1.5.6 Pointer subtraction

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

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

If the pointers point to objects whose size is no greater than four bytes, the alignment of data ensures that the division will be exact in all cases. For longer types, such as doubles and structures, the division may not be exact unless both pointers are to elements of the same array. Moreover, the quotient may be rounded up or down at different times, leading to potential inconsistencies.

1.5.7 Arithmetic operations

The compiler performs the usual arithmetic conversions set out in the ANSI C standard. 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 which arise naturally from two's complement representation.
- Right shifts on signed quantities are arithmetic.
- Any quantity which specifies the amount of a shift is treated as an unsigned 8-bit value.
- Any value to be shifted is treated as a 32-bit value.
- Left shifts of more than 31 give a result of zero.
- Right shifts of more than 31 give a result of zero from a shift of an unsigned or positive signed value; they yield -1 from a shift of a negative signed value.
- The remainder on integer division has the same sign as the divisor.
- If a value of integral type is truncated to a shorter signed integral type, the result is obtained as if by masking the original value to the length of the destination, and then sign extending.
- A conversion between integral types never causes an exception to be raised.
- Integer overflow does not raise an exception.
- Integer division by zero raises an exception.

By default, the following points apply to operations on floating-point types:

- When a double or long double is converted to a float, rounding is to the nearest representable value.
- A conversion from a floating type to an integral type causes an exception to be raised only if the value cannot be represented in a long int, (or unsigned long int in the case of conversion to an unsigned int).
- Floating-point underflow is not detected; any operation which underflows returns zero.
- Floating-point overflow raises an exception.
- Floating-point divide by zero raises an exception.

1.5.8 Expression evaluation

The compiler performs the usual arithmetic conversions (*promotions*) set out in the ANSI 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`.
- Between sequence points, the compiler may evaluate expressions in any order, regardless of parentheses. Thus the side effects of expressions between sequence points may occur in any order.
- Similarly, the compiler may evaluate function arguments in any order.

Any detail of order of evaluation not prescribed by the ANSI C standard may vary between releases of the ARM C compiler.

1.5.9 Implementation limits

The ANSI C standard sets out certain minimum limits which a conforming compiler must accept. You should be aware of these when porting applications between compilers. A summary is given ***Table 1-3: Implementation limits*** on page 1-25. The `mem` limit indicates that no limit is imposed by the ARM C compiler other than that imposed by the availability of memory.

Description	Required	ARM C
Nesting levels of compound statements and iteration / selection control structures.	15	mem
Nesting levels of conditional compilation.	8	mem
Declarators modifying a basic type.	31	mem
Expressions nested by parentheses.	32	mem
Significant characters: in internal identifiers and macro names in external identifiers	 31 6	 256 256
External identifiers in one source file.	511	mem
Identifiers with block scope in one block.	127	mem
Macro identifiers in one source file.	1024	mem
Parameters in one function definition / call.	31	mem

Table 1-3: Implementation limits

Description	Required	ARM C
Parameters in one macro definition / invocation.	31	mem
Characters in one logical source line.	509	no limit
Characters in a string literal.	509	mem
Bytes in a single object.	32767	mem [see Note]
Nesting levels for #included files.	8	mem
Case labels in a switch statement.	257	mem
Members in a single struct or union, enumeration constants in a single enum.	127	mem
Nesting of struct / union in a single declaration.	15	mem

Table 1-3: Implementation limits (Continued)

Note: *When running on 16-bit hosts, the ARM C compiler may impose a limit on the size of an object file. Generally, this limit will be 65535 bytes in a single object file rather than 32767 bytes in a single C-language object. 32-bit hosted versions do not have this limit.*

1.6 Standard Implementation Definition

Appendix A.6 of the ISO C standard collects together information about portability issues; section A.6.3 lists those points which must be defined by each implementation. This section corresponds to appendix A.6.3, dealing with aspects of the ARM C compiler and ANSI C library that are not defined by the ISO C standard, and which are implementation-defined.

1.6.1 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 was able to recover by guessing the user's intentions
`Serious error`	a violation of the ANSI specification from which no recovery was possible; the intention was not clear
`Fatal`	an indication that the compiler's limits have been exceeded or that the compiler has detected a fault in itself (for example, not enough memory)

1.6.2 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`*

1.6.3 Identifiers

256 characters are significant in identifiers with or without external linkage. Allowed characters are letters, digits, and underscores.

Case distinctions are significant in identifiers with external linkage.

In pcc mode (`-pcc` option) and limited pcc or system programming mode (`-fc` option), the dollar character ($) is also valid in identifiers.

1.6.4 Integers

The representations and sets of values of the integral types are set out in the *Software Development Toolkit User Guide (ARM DUI 0040).*

Note also:

- The result of converting an integer to a shorter signed integer (if the value cannot be represented) is as if the bits in the original value which cannot be represented in the final value are masked out, and the resulting integer sign-extended. The same applies when an unsigned integer is converted to a signed integer of equal length.
- Bitwise operations on signed integers yield the expected result given two's complement representation. No sign extension takes place.
- The sign of the remainder on integer division is the same as defined for the function `div()`.
- Right shift operations on signed integral types are arithmetic.

1.6.5 Characters

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.

Other properties of the source character set are host-specific, except that the ARM C compiler has no support for multi-byte character sets.

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

- The execution character set is identical to the source 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 NUL (or "\0") character.
- There are eight bits in a character in the execution character set.

- 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 which may be violated in any given retargeting of the generic ARM C library).
- Characters of the source character set in string literals and character constants map identically into the execution character set (by an assumption which may be violated in any given retargeting of the generic ARM C library).
- No locale is used to convert multi-byte characters into the corresponding wide characters (codes) for a wide character constant (not relevant to the generic implementation).
- A plain char is treated as unsigned (but as signed in pcc mode).

The character escape codes are shown in ***Table 1-4: 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
`\v`	`13`	Vertical tab
`\xnn`	`0xnn`	ASCII code in hexadecimal
`\nnn`	`0nnn`	ASCII code in octal

Table 1-4: Escape codes

1.6.6 Floating-point types

The representations and ranges of values of the floating-point types are given in the *Software Development Toolkit User Guide (ARM DUI 0040)*.

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

1.6.7 Arrays and pointers

The ISO standard specifies three areas in which the behavior of arrays and pointers must be documented. The points to note here are:

- The type `size_t` is unsigned int (signed int in PCC mode)
- Casting pointers to integers and vice versa involves no change of representation
- The type `ptrdiff_t` is defined as (signed int)

1.6.8 Registers

Using the ARM C compiler, you can declare any number of objects to have the storage class `register`. Depending on which variant of the ARM Procedure Call Standard (APCS) is in use, there are between five and seven registers available. Declaring more than this number of objects with register storage class must result in at least one of them not being held in a register. In general, it is advisable to declare no more than four.

The valid types are:

- any integer type
- any pointer type
- any integer-like structure (any one word struct or union in which all addressable fields have the same address or any one word structure containing only bitfields)
- a floating-point type, if software floating-point is used

Other variables, not declared with the register storage class, may be held in registers for extended periods, and register variables may be held in memory for some periods.

The double-precision floating type `double` occupies *two* ARM registers.

There is a `#pragma` which assigns a file-scope variable to a specified register everywhere within a compilation unit (see page 1-49).

1.6.9 Qualifiers

An object that has volatile-qualified type is *accessed* if any word or byte 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.

1.6.10 Declarators

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

1.6.11 Statements

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

1.6.12 Structure packing

Non-packed structs

By default, structures are aligned on word boundaries. Characters are aligned in bytes, shorts are aligned on even-numbered byte boundaries, and all other types, except bitfields, are aligned on word boundaries. Bitfields are subfields of ints, themselves aligned on word boundaries.

Structures may contain internal padding to ensure:

- members are correctly aligned
- the structure occupies a whole number of words

An example of a conventional (non-packed) struct is given in ***Figure 1-1: Conventional (non-packed) struct example***.

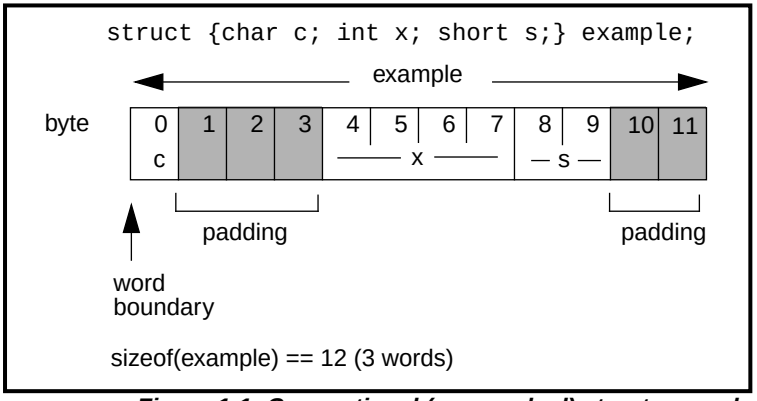


Figure 1-1: Conventional (non-packed) struct example

Packed structs

A packed struct is one in which there is neither padding between fields to ensure the natural alignment of each field, nor trailing padding to ensure the natural alignment of a following struct within an array.

Many applications read data from and write data to networks and computer buses in formats defined by international standards and by other programs executing on different processors. The data format is fixed. Data read and data to be written can be precisely mapped in C using packed structs. However, packed structs cannot support reading values of the wrong endianness.

On the ARM, access to unaligned data can be expensive (taking up to seven instructions and two extra work registers). Data accesses via packed structs should be minimized to avoid performance loss. Generally, internal data structures should not be padded.

Usage

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

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

```
#define packed __packed
```

`__packed` behaves as a type qualifier (like `volatile`) and may qualify any non-floating-point type.

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

Floating types may not be fields of packed structures.

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

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

As a result, the following will be faulted:

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

or

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

This ensures that a packed struct can never be assignment-compatible with an unpacked struct. This could happen if `__packed` were merely a type qualifier like `volatile` and `const`.

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

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

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

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

All top-level objects (global or local) are word-aligned and occupy an integral number of words of store, so there may be padding between separately declared top-level packed structs.

Sub-structs of packed structs

A struct (or union) sub-field of a packed struct or union must be declared to have packed struct (or packed union) type.

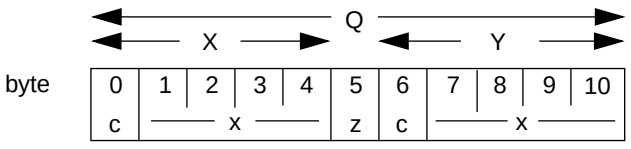
```
struct S {...};
__packed struct P {...};

struct T {
       struct S ss; /* OK */
       struct P pp; /* OK */
};

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

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

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



Note: The structure contains no padding.

Figure 1-2: Sub-struct padding

1.6.13 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.

1.6.14 Enumerations

An object of type `enum` is normally 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

This feature can reduce the size of the data area.
The command-line flag `-fy` sets the underlying type of `enum` to signed int.

1.6.15 Bitfields

The ARM C compiler handles bitfields in the following way:

- a plain bitfield (declared without either signed or unsigned qualifiers) is treated as unsigned (signed int in pcc mode) and has the same signedness as plain char.
- a bitfield which does not fit in a correctly aligned object of its type, is placed in the next such object.
- the order of allocation of bitfields within ints means that the first field specified occupies the lowest-addressed bits of the word
- bitfields do not straddle storage unit (int) boundaries

1.6.16 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 rules for directory searching are given in ***1.3.5 Included files*** on page 1-5. 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. See ***1.3.1 Naming conventions*** on page 1-4 for more details.

The recognized `#pragma` directives are shown in ***1.9.1 Pragmas*** on page 1-46.

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

1.6.17 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`*

These functions usually test only for characters whose values are in the range 0 to 127 (inc):

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

Characters with values greater than 127 return a result of 0 for all of 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 following statements also apply to character codes and affect the results returned by the `ctype()` functions:

Code	Description
128-159	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

Table 1-5: Character codes

Mathematical functions

The mathematical functions return the following values on domain errors:

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`

Table 1-6: Mathematical functions

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 is as follows:

Signal	Description
SIGABRT	abort
SIGFPE	arithmetic exception
SIGILL	illegal instruction
SIGINT	attention request from user
SIGSEGV	bad memory access
SIGTERM	termination request
SIGSTAK	stack overflow

Table 1-7: Signal function signals

The default handling of all recognized signals is to print a diagnostic message and call `exit`. This default behavior applies at program start-up.

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 (but note that a given targeting of it may not have them):

- 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.
- 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 following 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`

Table 1-8: perror() messages

Unspecified characteristics: 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()`

1.7 C Language Extensions

None of these extensions is available if the compiler is restricted to compiling strict ANSI C (eg by `-strict`).

1.7.1 // comments

The character sequence `//` starts a comment, which is terminated by the next newline character (as in C++). It should be noted 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's terminator, therefore:

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

1.7.2 Long long

64-bit integer types are available through the type specifier, long long.

Long long int and unsigned long long int are integral types, behaving analogously to (unsigned) long with respect to the usual arithmetic conversions.

Integer constants may have a `LL` suffix to force the type of the constant to long long (if it will fit) or unsigned long long, and `LLU` (or `ULL`) to force to unsigned long long.

In addition, a plain integer constant is of type (unsigned) long long if its value is large enough. [This is a quiet change: without long long, 2147483648 has type unsigned long; with long long, it has type long long. Thus, the value of 2147483648 > -1 is 1 in strict ANSI C, 0 with long long].

Bitfields may be of type (unsigned) long long.

Enumerators may not have valuesthat are too large to be contained in a variable of type long. (For example, long long enumerators are not available.)

The controlling expression of a switch statement may not have (unsigned) long long type, and consequently, case labels must also have values which can be contained in a variable of type unsigned long.

Printf and scanf format specifiers may include a `ll`, to specify that the following conversion applies to an (unsigned) long long argument, as in `%lld`.

1.8 Inline Assembler

Assembly language can be used to achieve more efficient code or to use features of the target processor which the compiler cannot use. The inline assembler enables the use of assembler instructions within a C program. This is useful in cases where a limited amount of assembler code is needed.

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

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

The tcc inline assembler implements the full Thumb instruction set.

1.8.1 Syntax

The assembler command is started by the assembler specifier `__asm`, and is followed by a list of assembler instructions inside braces. For example:

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

An instruction may optionally be followed by another instruction on the same line, separated by a semicolon. No separation is neccesary if instructions are on different lines.

You can use standard C/C++ comments anywhere within the inline assembler block. You can use the assembler command at any place inside a function where a C command is allowed.

1.8.2 Assembler instruction set

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

Operand expressions

Any register or constant operand may be an arbitrary C expression so that C variables may be read or written. The compiler adds any extra code to evaluate the expressions and allocates them to registers.

When an operand is written, the expression must be assignable (lvalue).

When writing code which uses both physical registers and C expressions, take care that you do not use too many registers, or the compiler may be unable to evaluate the expressions.

The compiler issues an error message if it detects a register allocation problem.

Constants

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

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

Instruction expansion

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

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

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

Labels

Labels have the same syntax as C labels:

`labelname:`

Labels can only be used by branch instructions in the form:

`B<cond> label`

Storage declarations

All storage can be declared or allocated in C and passed onto the inline assembler using C variables. Therefore, no additonal storage declarations are implemented.

Pseudo instructions

The pseudo instruction is NOP (no operation).

Note *The pseudo instructions ADR and ADRL are not implemented. (*`MOV expr1 & expr2` *can be used instead).*

Function calls

Calls using SWI or BL must specify exactly which calling standard is used. After the normal instruction fields, three optional register lists specify:

- the input parameters
- the registers which are output parameters after return
- the registers which corrupted by the called function

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

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

An omitted list is assumed to be empty, except for BL, which always corrupts LR.

The register lists have the same syntax as LDM/STM register lists.

The condition code register can be specified as PSR.

1.8.3 Examples

64-bit addition

```
typedef struct { unsigned int hi, lo; } uint64;
__value_in_regs uint64 addu64(uint64 a, uint64 b)
{
   uint64 res;
   __asm
   {
   ADDS    res.lo, a.lo, b.lo
   ADC     res.hi, a.hi, b.hi
   }
   return res;
}
```

String copying

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

Function call

```
int int_sum(int a, int b)     // uses R0 and R1, returns R0
{
   return a + b;
}
int sum_array(int arr[], int n)
{
   int sum = 0;
   while (--n >= 0)
        __asm
        {
              MOV  R0, sum
              MOV  R1, arr[n]
              BL   int_sum, {R0, R1}, {R0}, {R1,LR}
              MOV  sum, R0
        }
   return sum;
}
```

1.8.4 Pitfalls

1 C expressions with the comma operator must be bracketed:

```
ADD x, y, (f(), z)
```

2 The & operator cannot be used to denote hexadecimal constants. Use the C `0x` prefix instead:

```
AND x, y, 0xF00
```

3 When using physical registers, make sure that the compiler does not corrupt these registers when evaluating expressions. For example:

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

Because the compiler uses a function call to evaluate `x/y` (which corrupts R2,R3,IP,LR,PSR and alters R0 and R1), the value in R0 has been lost.

The compiler can detect the corruption in many cases; for example when it needs a temporary register, but the register is already in use:

```
__asm
{
   MOV    ip, #3
   ADDS   x, x, #0x12345678// instruction is expanded
   ORRCS  x, x, ip
}
```

Here, the compiler uses IP as a temporary to expand the ADD instruction, corrupting the value 3 in IP. This results in an error.

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

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

This code returns x unaltered. In fact, the compiler assumes that x and R0 are two different variables, despite the fact that x is allocated to R0 on both function entry and exit. As the assembler code doesn't do anything useful, it is optimized away.

1.8.5 Restrictions

- You cannot write PC, SP, FP, SL and SB (where applicable, depending on the selected calling standard). Other registers, like IP, LR, R0-R3, PSR must be used with caution, as these may be used as temporary registers by the compiler when evaluating expressions.
- LDM/STM instructions currently only allow physical registers to be specified in the register list.
- BX is not yet implemented.
- Changing processor modes, or altering the state of coprocessors is permitted, but the compiler is unaware of this.
  - After changing to a different processor mode, no C expressions are allowed until the mode is changed back to the original mode.
  - Equally, when changing state of a FP coprocessor by executing FP instructions, no floating-point expressions may be used until the original state has been restored.

1.9 Compiler-specific Features

1.9.1 Pragmas

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

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

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

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

The list of recognized pragmas is shown in ***Table 1-9: Pragmas*** on page 1-47.

1.9.2 Specifying pragmas from the command line

Any pragma can also be specified from the compiler's command line using:

```
-zpLetterDigit
```

1.9.3 Pragmas controlling the preprocessor

`continue_after_hash_error`	Implements a `#warning "..."` preprocessor directive.
`include_only_once`	The containing `#include` file is included only once, and if its name recurs in a subsequent `#include` directive, the directive is ignored.
`force_top_level`	The containing `#include` file should only be included at the top level of a file. A syntax error results if the file is included within a function.

Values marked with * are the default values.

Thumb The options marked with † are not available in Thumb.

	Pragma name	Short Form		'no' Form	
†	check_memory_accesses	c1		c0	*
	check_printf_formats	v1		v0	*
	check_scanf_formats	v2		v0	*
	check_stack [1]	s0	*	s1	
	continue_after_hash_error	e1		e0	*
	force_top_level	t1		t0	*
†	FP register variable	f1-f4		f0	*
	include_only_once	i1		i0	*
	integer register variable	r1-r7y		r0	*
	optimise_crossjump	j1	*	j0	
	optimise_cse	z1	*	z0	
	optimise_multiple_loads	m1	*	m0	
†	profile	p1		p0	*
†	profile_statements	p2		p0	*
	side_effects	y0	*	y1	
	warn_deprecated	d1	*	d0	
	warn_implicit_fn_decls	a1	*	a0	

Table 1-9: Pragmas

1. s0 is the default for ARM. Thumb defaults to s1.

1.9.4 Pragmas controlling printf/scanf argument checking

Pragmas `check_printf_formats` and `check_scanf_formats` control whether the actual arguments to `printf` and `scanf` are type-checked against the format designators in a literal format string. Calls using non-literal format strings cannot be checked. By default, all calls involving literal format strings are checked.

1.9.5 Pragmas controlling optimization

Pragmas `optimise_crossjump`, `optimise_multiple_loads` and `optimise_cse` give fine control over where these optimizations are applied. For example, it is sometimes advantageous to disable cross-jumping (the “common tail” optimization) in the critical loop of an interpreter; and it may be helpful in a timing loop to disable common sub-expression elimination and the optimization of multiple load instructions to load multiples. Note that correct use of the `volatile` qualifier should remove most of the more obvious needs for this degree of control (and `volatile` is also available in the ARM C compiler's pcc mode unless `-strict` is specified).

By default, functions are assumed to be impure, so function invocations are not candidates for common sub-expression elimination. The `noside_effects` pragma asserts that the function declarations up to the next `#pragma side_effects` describe pure functions, invocations of which can be CSEs. See also ***__pure*** on page 1-50.

1.9.6 Pragmas controlling code generation

Stack-limit checking (ARM processors only)

If the compiler is configured to compile code for the explicit stack limit variant of the ARM Procedure Call Standard (documented in ***Chapter 5, ARM Procedure Call Standard***), `#pragma nocheck_stack` disables the generation of code at function entry which checks for stack limit violation. There is little advantage in turning off this check: it typically costs only two instructions and two machine cycles per function call.

Note *You must use* `nocheck_stack` *in writing a signal handler for the* **SIGSTAK** *event. When this occurs, stack overflow has already been detected, so checking for it again in the handler results in a fatal circular recursion.*

Memory access checking

The pragma `check_memory_accesses` 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 1,2,4 for byte, short, long writes, respectively.
It is up to your library implementation to check that the address given is reasonable.

Global (program-wide) register variables

The pragmas `f0`–`f4` and `r0`–`r7` have no long form counterparts. Each introduces or terminates a list of extern, file-scope variable declarations. Each such declaration declares a name for the *same* register variable.

For example:

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

Any type that can be allocated to a register (see ***1.6.8 Registers*** on page 1-31) can be allocated to a global register. Similarly, any floating-point type can be allocated to a floating-point register variable.

Global register r1 is the same as register v1 in the ARM Procedure Call Standard (APCS); similarly r2 equates to v2, and so on. Depending on the APCS variant, between five and seven integer registers (v1–v7, machine registers r4–r10) and four floating-point registers (f4–f7) are available as register variables. In practice, it is probably unwise to use more than three global integer register variable and two global floating-point register variables.

Provided the same declarations are made in each separate compilation unit, a global register variable may exist program-wide.

Otherwise, 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 which does not use it as a global register variable, such as a library function.

A corollary of the safety of direct calls out of a global-register-using compilation unit is that calls back into it are dangerous. In particular, a global-register-using function called from a compilation unit, which uses that register as a compiler-allocated register, will probably read the wrong values from its supposed global register variables.

Currently, there is no check at link-time to ensure that direct calls are sensible. And even if there were, indirect calls via function arguments pose a hazard which is harder to detect. This facility must be used with care. Preferably, the declaration of the global register variable should be made in each compilation unit of the program. See also ***__global_reg(n)*** on page 1-51.

1.9.7 Function declaration keywords

Several function declaration options tell the compiler to give a function special treatment.

Note *None of these keywords are portable to other C compilers.*

`__inline`

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

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

Currently, the compiler always inlines functions when `__inline` is used. Code density and performance could be adversely affected if large functions are inlined.

`__irq`

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

`__pure`

By default, functions are assumed to be impure (ie. they have side-effects), so function invocations are not candidates for common subexpression elimination. `__pure` has the same effect as pragma `noside_effects`, and asserts that the function declared is a pure function, invocations of which can be CSEs.

Note: *A function is only properly defined as pure if its result depends only on the value of its scalar argument. This means that pointers may not be passed into pure functions because the compiler cannot tell how much they point at.*

`__swi`
`__swi_indirect`

A SWI taking up to four arguments (in registers 0 to argcount –1) and returning up to four results (in registers 0 to resultcount–1) can be described by a C function declaration, which causes uses of the function to be compiled inline as a SWI.

For a SWI returning 0 results use:

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

For example:

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

For a SWI returning 1 result, use:

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

For a SWI returning more than 1 result:

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

Note: `__value_in_regs` *is needed to specify that a (short) structure value is returned in registers, rather than by the usual indirection mechanism specified in the ARM Procedure Call Standard.*

If there is an indirect SWI (taking the number of the SWI to call as an argument in r12), calls through this SWI can similarly be described by a C function declaration such as:

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

For example:

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

This might be called as:

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

`__value_in_regs`

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

```
typedef struct int64_structt {
unsigned int lo;
unsigned int hi;
} int64;
__value_in_regs extern int64 mul64(unsigned a, unsigned b);
```

See ***Chapter 5, ARM Procedure Call Standard*** for information on the default method of passing and returning structures.

1.9.8 Variable declaration keywords

`__global_reg(n)`

Allocates the declared variable to a global integer register variable, in the same way as `#pragma rn`. The variable must have an integral or pointer type. See also ***1.9.6 Pragmas controlling code generation*** on page 1-48.

`__global_freg(n)`

Allocates the declared variable to a global floating-point register variable, in the same way as `#pragma fn`. The variable must have type float or double. See also ***1.9.6 Pragmas controlling code generation*** on page 1-48.

Note: *The global register, whether specified by keyword or pragmas, must be specified in all declarations of the same variable. So, the following is an error:*

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

1.9.9 Predefined macros

In the table below, where the value field is empty, the symbol concerned is merely defined, as though by (for example) `-D__arm` on the command line.

Name	Value	Notes
`__STDC__`	1	except in pcc mode not defined in pcc mode
`__arm`		defined if using armcc or tcc
`__thumb`		defined if using tcc
`__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)
`__PCS_FPREGARGS`		defined if apcs `/fpregargs` is in use
`__BIG_ENDIAN`		defined if compiling for a big-endian target
`__TARGET_ARCH_`*xx*		*xx* represents the target architecture. So, for example, using `-arch 4T`, `__TARGET_ARCH_4T` is defined, and no other symbol starting with `__TARGET_ARCH_` is defined.
`__TARGET_CPU_`*xx*		*xx* represents the target cpu. So, for example, with `-cpu ARM7TM`, `__TARGET_CPU_ARM7TM` is defined, and no other symbol starting `__TARGET_CPU_` is defined. If there is no target cpu, merely a target architecture, the `__TARGET_CPU_generic` is defined.
`__TARGET_FEATURE_HALFWORD`		defined if the target architecture supports halfword and signed byte access instructions.

Table 1-10: Predefined macros

Name	Value	Notes
`__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. In release 2.11, the value is 4.69.
`__CLK_TCK`	100	centisecond clock definition
`__CC_NORCROFT`	1	set by all Codemist compilers
`__sizeof_int`	4	`sizeof(int)`, but available in preprocessor expressions
`__sizeof_long`	4	`sizeof(long)`, but available in preprocessor expressions
`__sizeof_ptr`	4	`sizeof(void *)`, but available in preprocessor expressions

Table 1-10: Predefined macros

2 Assembler

This chapter describes the ARM Assembler. If you only need to understand the assembly language output by the C compilers, see the datasheet for your device.

2.1 Overview

The ARM assembler is a two-pass assembler, processing its source files twice to reduce the amount of internal state that it needs to keep.

The ARM assembler, armasm, compiles ARM assembly language into ARM Object Format object code. This code can then be linked with object code produced by the ARM assembler or the ARM C compiler, and with object libraries created by the ARM librarian.

The Thumb assembler, tasm, compiles both ARM and Thumb assembly language into ARM Object Format object code.

2.2 Command Syntax

The command to invoke the ARM assemblers is one of:

```
armasm {options} sourcefile objectfile
tasm {options} -o objectfile sourcefile
```

2.2.1 Command-line options

The options are listed below. Permitted abbreviations are shown underlined.

`-apcs` *option*`{/`*qualifier*`}{/`*qualifier*`...}`

Specifies whether the ARM Procedure Call Standard (APCS) is in use, and also specifies some attributes of CODE AREAs:

`NONE`	No APCS registers set up
3	APCS version 3 registers set up

Qualifiers should only be used with 3.
See also ***Predeclared register names*** on page 2-4.

`-arch` *architecture*

Sets the target architecture. Legitimate values are:

- 3
- 3M
- 4
- 4T

Some processor-specific instructions produce either errors or warnings if assembled for the wrong target architecture.

`-bigend`

Assembles code suitable for a big-endian ARM, (by setting the built-in variable `{ENDIAN}` to `big`).

`-checkreglist`

Checks 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. This can be used to help detect misuse of symbolic register names.

`-cpu` *`ARMcore`*
: Sets the target ARM core. Legitimate values include:
  - `ARM6`
  - `ARM7`
  - `ARM7M`
  - `ARM7TM`
  - `ARM8`
  - `StrongARM`

  Some processor-specific instructions will produce warnings if assembled for the wrong ARM core.

`-depend` *`dependfile`*
: Saves source file dependency lists, which are suitable for use with make utilities.

`-errors` *`errorfile`*
: Outputs error messages to *`errorfile`*.

`-g`
: Outputs ARM Symbolic Debugger debugging tables, suitable for use with armsd and the ARM Debugger for Windows.

`-help`
: Displays a summary of the command-line options.

`-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. The search rule used is similar to the ANSI C search rule; the current place is the directory where the current file was found.

`-list` *`listingfile`*
: Several options work with `-list`:

  `-noterse`
  : Turns the `terse` flag off. When the `terse` flag is on, lines skipped due to conditional assembly do not appear in the listing. With the `terse` flag off, these lines appear in the listing. The default is on.

  `-width` *`n`*
  : Sets the listing page width. (Default is 79.)

  `-length` *`n`*
  : Sets the listing page length. Length zero means an unpaged listing. (Default is 66.)

  `-xref`
  : Lists cross-referencing information on symbols: where they were defined; and where they were used, both inside and outside macros. The default is off.

	`-littleend`	Assembles code suitable for a little-endian ARM, (by setting the built-in variable `{ENDIAN}` to `little`).
	`-maxcache` *n*	Sets the maximum source cache size. The default is 8MB.
	`-nocache`	Turns off source caching. Source caching is performed when reading source files on the first pass, so that they can be read from memory during the second pass. The default is on.
	`-noesc`	Ignores C-style special characters (`\n`, `\t`, etc.).
	`-noregs`	Tells the assembler not to predefine implicit register names (r0–r15, f0–f7, a1–a4, v1–v6, sl, fp, ip, sp, lr, pc).
	`-nowarn`	Turns off warning messages.
	`-predefine` *directive*	Pre-executes a `SETx` directive. This implicitly executes a corresponding `GBLx` directive. The full `SETx` argument must be quoted as it contains spaces, for example: `-pd "Version SETA 44"`
	`-unsafe`	Changes into warnings any errors produced due to the selected architecture and cpu.
	`-via` *file*	Opens *file* and reads in more armasm command-line arguments. This is intended mainly for hosts such as a PC, where command-line length is severely limited.
Thumb	`-16`	Tells the assembler to interpret instructions as Thumb instructions.This is equivalent to placing a `CODE16` directive at the head of the source file.
	`-32`	Tells the assembler to interpret instructions as ARM instructions.

Predeclared register names

By default the following register names are predeclared:

- R0–R15
- r0–r15
- sp and SP
- lr and LR
- pc and PC

If the APCS is in use, the following register names are also predeclared:

- a1–a4
- v1–v6
- sl
- fp, ip, and sp

Qualifiers

The qualifiers are as follows:

`/reentrant` Sets the re-entrant attribute for any code AREAs, and predeclares sb (static base) in place of v6.

`/32bit` Informs the linker that the code being generated is written for 32-bit ARMs. The armasm built-in variable `{CONFIG}` is also set to 32. This is the default setting.

`/26bit` Tells the linker that the code is intended for 26-bit ARMs. The armasm built-in variable `{CONFIG}` is also set to 26. Note that these options do not themselves generate particular ARM-specific code, but allow the linker to warn of any mismatch between files being linked, and also allow programs to use the standard built-in variable `{CONFIG}` to determine the code to produce.

`/swstackcheck` Marks CODE AREAs as using sl for the stack limit register, following the APCS (the default setting).

`/noswstackcheck` Marks CODE AREAs as not using software stack-limit checking, and predeclares an additional v-register:

v6	if re-entrant
v7	if not re-entrant

2.3 Assembly Language Overview

Assembly language is the language which the assembler parses and compiles to produce object code in ARM Object Format. This can be:

- ARM assembly language
- Thumb assembly language
- a mixture of both

This section deals with features that are common to both ARM and Thumb assembly language. For language-specific information, see the *ARM Architecture Reference Manual (ARM DDI 0100)*, which describes every ARM and THUMB instruction in terms of syntax and availability. Refer to the *Software Development Toolkit User Guide (ARM DUI 0040)* for information on writing assembly language modules.

2.3.1 Case rules

Instruction mnemonics and register names may be written in uppercase or lowercase, but not mixed. Directives must be written in uppercase.

2.3.2 Input lines

The general form of assembler input lines is:

```
{label} {instruction} {;comment}
```

A space or tab should separate the label (where one is used) and the instruction. If no label is used the line must begin with a space or tab. Any combination of these three items will produce a valid line; empty lines are also accepted by the assembler and can be used to improve the clarity of source code.

Line length

Assembler source lines may be up to 255 characters long.

To make source files easier to read, a long line of source can be split onto several lines by placing a backslash character, '\', at the end of a line. The backslash must not be followed by any other characters (including spaces or tabs). The backslash with the end-of-line sequence is treated by the assembler as white space.

Note *Do not use the backslash with the end-of-line sequence within quoted strings.*

2.3.3 AREAs

AREAs are independent, named, indivisible chunks of code and data manipulated by the linker. The linker places each AREA in a program image according to the AREA placement rules (ie. not necessarily adjacent to the AREAs with which it was assembled or compiled). Conventionally, the output of an assemble or compilation consists of two AREAs:

- one for the code (usually marked read-only)
- one for the data which may be written to

AREA syntax

The syntax of the AREA directive is:

```
AREA name{,attr}{,attr}...
```

You may choose any name for your AREAs, but certain choices 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. AREA attributes are as follows:

ABS	Is absolute (rooted at a fixed address).
ALIGN=*expression*	Forces the start of the AREA to be aligned on a power-of-two byte-address boundary. By default, AREAs are aligned on a 4-byte word boundary, but the expression can have any value between 2 and 32 inclusive.
BASED *Rn*	Is the static base data AREA containing tables of address constants locating static data items. *Rn* is a register, conventionally r9. Any label defined within this AREA becomes a register-relative expression which can be used with LDR and STR instructions. For full details see ***Chapter 5, ARM Procedure Call Standard***.
CODE	Contains machine instructions. READONLY is the default.
COMDEF	Is the common AREA definition.
COMMON	Is the common AREA.
DATA	Contains data, not instructions. READWRITE is the default.
HALFWORD	Indicates that the code AREA contains ARM halfword instructions.
INTERWORK	Indicates that the code AREA is suitable for ARM/Thumb interworking.
NOINIT	Indicates that data AREA is initialized to zero. It contains only space reservation directives, with no initialized values.
PIC	Indicates position-independent code. It will execute where loaded without modification.
READONLY	Indicates that this AREA will not be written to (default).
READWRITE	Indicates that this AREA may be read and written to.
REENTRANT	Indicates that the code AREA is re-entrant.
REL	Is relocatable. It may be relocated by the linker (default).

In ARM assembly language, each AREA begins with an AREA directive. If the directive is missing, the assembler generates an AREA with an unlikely name (|$$$$$$$|) and produces a diagnostic message. This limits the number of spurious errors caused by the missing directive, but does not lead to a successful assembly. A re-entrant object generally has a third AREA marked:

```
BASED sb
```

which will contain relocatable address constants. This allows the code area to be read-only, position-independent and re-entrant, making it easily ROM-able.

2.3.4 ORG and ABS

The ORG (origin) directive sets the base address and the ABS (absolute) attribute of the containing AREA, or of the following AREA if there is no containing AREA:

```
ORG base-address
```

In some circumstances, this creates objects which cannot be linked. In general, it only makes sense to use ORG in programs consisting of one AREA, to map fixed hardware addresses such as trap vector locations. Otherwise, ORG should be avoided.

2.3.5 Symbols

Numbers, logical values, string values and addresses may be represented by symbols. Symbols representing numbers or addresses, logical values and strings are declared using the GBL and LCL directives, and values are assigned immediately by SETA, SETL and SETS directives respectively (see section ***2.6.2 Local and global variables*** on page 2-23). Addresses are assigned by the assembler as assembly proceeds, some remaining in symbolic, relocatable form until link time.

- Symbols must start with an uppercase or lowercase letter; the assembler treats the two forms as distinct. Numeric characters and the underscore character may be part of the symbol name. All characters are significant.
- Symbols should not use the same name as instruction mnemonics or directives. While the assembler can distinguish between them through their relative positions in the input line, a programmer may not be able to do so.
- Symbol length is limited by the 255-character line-length limit.

Symbol name delimiters

If there is a need to use a wider range of characters in symbols—for instance when working with other compilers—use enclosing bars to delimit the symbol name; for example, |C$$code|. The bars are not part of the symbol.

2.3.6 Labels

Labels are a special form of symbol, distinguished by their position at the start of lines. The address given by a label is not explicitly stated, but is calculated during assembly.

2.3.7 Local labels

The local label is a subclass of label, and begins with a number in the range 0-99. Local labels work in conjunction with the ROUT directive and are useful for solving the problem of macro-generated labels. Unlike global labels, a local label may be defined many times; the assembler uses the definition closest to the point of reference.

Beginning a local area label

The label area starts with the next line of source, and ends with the next ROUT directive or the end of the program. To begin a local label area, use:

```
{label} ROUT
```

Defining local labels

Local labels are defined as:

```
number{routinename}
```

When defining a local label, *routinename* need not be used. If omitted, it is assumed to match the label of the last ROUT directive. It is an error to give a routine name when no label has been attached to the preceding ROUT directive.

Making a reference to a local label

The syntax is:

```
%{x}{y}n{routinename}
```

where:

%
: Introduces the reference and may be used anywhere where an ordinary label reference is valid.

x
: Tells the assembler where to search for the label:

  B — indicates backward

  F — indicates forward

  If no direction is specified, the assembler looks both forward and backward. However, searches will never go outside the local label area (that is, beyond the nearest ROUT directives).

y
: Provides the following options:

  A — looks at all macro levels

  T — looks only at this macro level

  If *y* is absent, the assembler looks at all macros from the current level to the top level.

n{*routinename*}
: Is the number of the local label. If *routinename* is present, it is checked against the enclosing ROUT label.

2.3.8 Comments

The first semicolon on a line marks the beginning of a comment, except where the semicolon appears inside a string constant. A comment alone is a valid line. All comments are ignored by the assembler.

2.3.9 Constants

Numbers	Numeric constants are accepted in three forms:		
	decimal	for example, `123`	
	hexadecimal	for example, `&7B`	
	n_xxx	where:	
		n	is a base between 2 and 9
		xxx	is a number in that base
Strings	Strings consist of opening and closing double quotes, enclosing characters and spaces. If double quotes or dollar signs are used within a string as literal text characters, they should be represented by a pair of the appropriate character; for example $$ for $. The standard C escape sequences can be used within string constants.		
Boolean	The Boolean constants “true” and “false” should be written as `{TRUE}` and `{FALSE}`.		
Characters	Character constants consist of opening and closing single quotes, enclosing either a single character or an “escaped” character, using the standard C escape characters.		

2.3.10 The END directive

Every assembly language source must end with:

```
END
```

on a line by itself.

2.3.11 Symbolic assembly

ARM instructions

`LDR` can be used 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. The syntax is:

```
LDR register,=expression
```

If *`expression`* is a numeric constant, a `MOV` or `MVN` will be used rather than an `LDR` if the constant can be constructed by either of these instructions. Otherwise, the assembler will generate a program-relative `LDR`, and if the desired literal does not already exist within the addressable range of this `LDR`, it will place the literal in the next literal pool, (see also LTORG in ***2.5.4 Organizational directives: END, ORG, LTORG and KEEP*** on page 2-19*)*.

Additionally, LDR or STR can be used to transfer data to or from an address specified by a label (optionally with an offset) as follows:

```
opcode{cond}{B} register,label-expression
```

When used in this form, *label-expression* must either be addressable PC-relative from this instruction, or must be a register-relative label created using the '^' directive with a register operand, (see ***2.5.3 Describing the layout of store: ^ and #*** on page 2-18).

THUMB instructions

The assembler also accepts the following forms:

```
LDR     source, label
LDR     source, =<expr>
```

where:

label is a program label defined within the addressable range of this instruction (ie. within the range +4 to +1024, allowing for the PC being offset from the current instruction by 4.

expr may be either:

- an expression evaluating to a numeric constant
- an external symbol, optionally + or - a numeric constant

The value of *expr* is placed in the next literal pool. If the same numeric constant is referenced more than once in a given literal pool, only one copy of the constant is placed in the literal pool. If an external symbol is used, a relocation directive will be placed in the object file to relocate the value in the literal pool by the value of the external symbol when the object file is linked.

2.3.12 Pseudo-instructions

The assemblers support several pseudo-instructions which are translated into the appropriate combination of ARM instructions at assembly time.

ARM instructions

ADR The pseudo-instruction ADR assembles address to register. Because the ARM has no "load effective address" instruction, the assembler provides ADR, which always assembles to produce ADD or SUB instructions to generate the address. The syntax is:

```
ADR{condition}{L} register,expression
```

The *expression* can be register-relative, program-relative or numeric. ADR must assemble to one instruction, whereas ADRL allows a wider range of effective addresses to be assembled in two instructions.

NOP — NOP generates the preferred no-operation code for a given ARM processor, which is:

```
MOV R0,R0.
```

NOP is really a directive and so cannot be used conditionally; not executing a no-operation is the same as executing it, so conditional execution would be pointless.

THUMB instructions

ADR — places address of label in reg:

```
ADR reg, label
```

label must be defined locally, it cannot be imported.

The range of ADR is limited: +4 to +1024 from the current instruction. label must be aligned.

MOV — has the syntax:

```
MOV Rd, Rs
```

If Rd and Rs are both low registers, a MOV instruction is synthesized using an ADD immediate instruction with a zero immediate value. This MOV Rd, Rs generates the opcode for ADD Rd, Rs, #0

This has the side effect of altering the condition codes.

NOP — The Thumb NOP pseudo instruction generates a MOV R8,R8 instruction.

```
NOP
```

The ARM NOP generates a MOV R0, R0 instruction. Hence, the condition codes are unaltered by ARM or Thumb NOPs.

2.4 Expressions and Operators

Expressions are combinations of simple 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.

Adjacent unary operators evaluate from right to left; 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 which resemble their counterparts in high-level languages.

2.4.1 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.

Operator	Usage	Explanation
?	?A	Number of bytes generated by line defining label A.
BASE INDEX	:BASE:A :INDEX:A	If A is a PC-relative or register-relative expression: BASE returns the number of its register component, and INDEX returns the offset from that base register. BASE and INDEX are most useful in macros.
+ and -	+A -A	Unary plus. Unary negate. + and – can act on numeric, program-relative and string expressions.
LEN	:LEN:A	Length of string A.
CHR	:CHR:A	ASCII string of 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}.

Table 2-1: Operator precedence

2.4.2 Binary operators

Binary operators are written between the pair of sub-expressions on which they operate. 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 which bind most tightly and have the highest precedence:

Operator	Usage	Explanation
*	A*B	multiply
/	A/B	divide
MOD	A:MOD:B	A modulo B

Table 2-2: Multiplicative operators

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.

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

Table 2-3: String manipulation operators

Shift operators

The shift operators act on numeric expressions, shifting or rotating the first operand by the amount specified by the second.

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

Table 2-4: Shift operators

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.

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

Table 2-5: Addition and logical operators

Relational operators

Relational operators act on two operands of the same type to produce a logical value:

- numeric
- program-relative
- register-relative
- strings

Strings are sorted using ASCII ordering. String A will be less than string B if it is either a leading substring of string B, or if the left-most character of A in which the two strings differ is less than the corresponding character in string B.

Note that arithmetic values are unsigned, so the value of `0>-1` is `{FALSE}`.

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

Table 2-6: Relational operators

Boolean operators

These are the weakest binding operators with the lowest precedence.

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

Table 2-7: Boolean operators

The Boolean operators perform the standard logical operations on their operands, which should evaluate to `{TRUE}` or `{FALSE}`.

2.5 Directives

2.5.1 Storage reservation and initialization: DCB, DCW and DCD

DCB	defines one or more bytes: can be replaced by =
DCW	defines one or more halfwords (16-bit numbers)
DCD	defines one or more words: can be replaced by &
%	reserves a zeroed area of store

The syntax of DCB, DCW, DCD is:

```
{label} directive expression-list
```

DCD can take program-relative and external expressions as well as numeric ones. In the case of DCB, the *expression-list* can include string expressions, whose characters are loaded into consecutive bytes in store. Unlike C strings, armasm strings do not contain an implicit trailing NUL, so a C string has to be fabricated as follows:

```
C_string DCB "C_string",0
```

The syntax of % is:

```
{label} % numeric-expression
```

This directive sets to zero the number of bytes specified by *numeric-expression*.

An external expression consists of an external symbol followed optionally by a constant expression. The external symbol must come first.

Thumb

DCD and Thumb

If you use the DCD directive with a Thumb label within a code area, the value stored is that of the Thumb label plus 1. This is because bit 0 of the register used in a BX instruction must be set to 1 in order to change state from ARM to Thumb. To avoid this, use the DATA directive when decoding data in code. See ***2.5.11 Thumb-specific directives: CODE 16, CODE32 and DATA*** on page 2-22 for details of the DATA directive.

2.5.2 Floating-point store initialization: DCFS and DCFD

DCFS	defines single-precision floating-point values
DCFD	defines double-precision floating-point values

The syntax of these directives is:

```
{label} directive fp-constant{,fp-constant}
```

Single-precision numbers occupy one word, and double-precision numbers occupy two; both should be word-aligned. An *fp-constant* takes one of the following forms:

```
{-}integer E{-}integer                  For example; 1E3, -4E-9
{-}{integer}.integer{E{-}integer}       For example; 1.0, -.1, 3.1E6
```

E may also be written in lowercase.

2.5.3 Describing the layout of store: ^ and

^ sets the origin of a storage map

reserves space within a storage map

The syntax of these directives is:

```
            ^ expression{,base-register}
{label}     # expression
```

where:

^ This directive sets the origin of a storage map at the address specified by *expression*. A storage-map location counter, @, is also set to the same address. The *expression* must be fully evaluable in the first pass of the assembly, but may be program-relative. If no ^ directive is used, the @ counter is set to zero; it can be reset any number of times using ^ to allow many storage maps to be established.

Space within a storage map is described by the # directive. Every time # is used its *label* (if any) is given the value of the storage location counter @, and @ is then incremented by the number of bytes reserved.

In a ^ directive with a *base-register*, the register becomes implicit in all symbols defined by # directives which follow, until cancelled by a subsequent ^ directive.
These register-relative symbols can later be quoted in load and store instructions.

For example:

```
    ^ 0,r9
    # 4
Lab # 4
    LDR   r0,Lab
```

is equivalent to:

```
LDR   r0,[r9,#4]
```

2.5.4 Organizational directives: END, ORG, LTORG and KEEP

`END` Stops the processing of a source file. If assembly of the file was invoked by a `GET` directive, the assembler returns and continues after the `GET` directive (see ***2.5.6 Links to other source files: GET/INCLUDE*** on page 2-20). If `END` is reached in the top-level source file during the first pass without any errors, the second pass begins. No `END` directive is an error.

`ORG numeric-expression`

Determines a program's origin; it sets the initial value of the program location counter. Only one `ORG` is allowed in an assembly and no ARM instructions or store initialization directives may precede it. If there is no `ORG`, the program is relocatable and the program counter is initialized to 0.

`LTORG` Directs the current literal pool to be assembled immediately following it. A default `LTORG` is executed at every `END` directive which is not part of a nested assembly. Large programs may need several literal pools, each closer to their literals' location to avoid violating LDR's 4KB offset limit.

`KEEP {symbol}`

Retains local (non-exported) symbols in the assembler's symbol table. The assembler does not by default describe local (non-exported) symbols in its output object file (see ***2.5.5 Links to other object files: IMPORT and EXPORT***). If the `KEEP` directive is used alone, all symbols are kept; if only a specific symbol needs to be kept it can be specified by name.

2.5.5 Links to other object files: IMPORT and EXPORT

`IMPORT` *`symbol`*`{[FPREGARGS]}{,WEAK}`

Provides the assembler with a name (*`symbol`*) which is not defined in this assembly, but which is resolved at link time to a symbol defined in a separate object file. The symbol is treated as a program address; if the `WEAK` attribute is given, the linker does not fault an unresolved reference to this symbol, but zeroes the location referring to it. If `[FPREGARGS]` is present, the symbol defines a function which expects floating-point arguments passed in floating-point registers.

`EXPORT` *`symbol`*`{[FPREGARGS,DATA,LEAF]}`

Declares a symbol for use at link time by other, separate object files.

`FPREGARGS`	defines a function which expects floating-point arguments to be passed in floating-point registers
`DATA`	defines a code-segment datum rather than a function or a procedure entry point
`LEAF`	denotes that it is a leaf function which calls no other functions

2.5.6 Links to other source files: GET/INCLUDE

`GET` *`filename`*

Includes a file within the file being assembled. This may in turn use `GET` directives to include more files. Once assembly of the included file is complete, assembly continues at the line following the `GET` directive.

`INCLUDE` *`filename`*

Is a synonym for `GET`.

2.5.7 Diagnostic generation: ASSERT, !, and INFO

`ASSERT` *`logical-expression`*

Supports diagnostic generation. If the *`logical-expression`* returns `{FALSE}`, a diagnostic is generated during the second pass of the assembly. `ASSERT` can be used both inside and outside macros.

`!` *`arithmetic-expression, string-expression`*

Is related to `ASSERT`, but is inspected on both passes of the assembly, providing a more flexible means for creating custom error messages. The arithmetic expression is evaluated; if it equals zero, no action is taken during pass one, but the string is printed as a warning during pass two. If the expression does not equal zero, the string is printed as a diagnostic and the assembly halts after pass one.

`INFO` *`arithmetic-expression, string-expression`*

The arithmetic expression is evaluated. It it equals zero, no action is taken during pass one, but the string is prefixed with the source file and line number, and is printed as a warning during pass two. If the expression does not equal zero, the string is printed as a diagnostic and the assembly halts after pass one.

2.5.8 Titles: TTL and SUBT

Titles can be specified within the code using the `TTL` (title) and `SUBT` (subtitle) directives. Each is used on all pages until a new title or subtitle is called.

If more than one appears on a page, only the latest will be used: the directives alone create blank lines at the top of the page.

The syntax is:

```
TTL title
SUBT subtitle
```

2.5.9 Dynamic listing options: OPT

The `OPT` directive is used to set listing options from within the source code, providing that listing is turned on.

The default setting is to produce a normal listing including the declaration of variables, macro expansions, call-conditioned directives and `MEND` directives, but without producing a listing during the first pass.

These settings can be altered by adding the appropriate values from the list, and using them with the OPT directive as shown in ***Table 2-8: 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 the listing of SET, GBL and LCL directives.
32	Turns off the listing of SET, GBL and LCL directives.
64	Turns on the listing of macro expansions.
128	Turns off the listing of macro expansions.
256	Turns on the listing of macro calls.
512	Turns off the listing of macro calls.
1024	Turns on the first pass listing.
2048	Turns off the first pass listing.
4096	Turns on the listing of conditional directives.
8192	Turns off the listing of conditional directives.
16384	Turns on the listing of MEND directives.
32768	Turns off the listing of MEND directives.

Table 2-8: OPT directive settings

2.5.10 Miscellaneous directives: ALIGN, NOFP, RLIST and ENTRY

`ALIGN` {*power-of-two*{,*offset-expression*}}

After store-loading directives have been used, the program counter (PC) will not necessarily point to a word boundary. If an instruction mnemonic is encountered, the assembler inserts up to three bytes of zeros to achieve alignment. However, an intervening label may not then address the following instruction. If this label is required, `ALIGN` should be used. On its own, `ALIGN` sets the instruction location to the next word boundary; the optional *power-of-two* parameter can be used to align with a coarser byte boundary, and the *offset-expression* parameter to define a byte offset from that boundary.

`NOFP` In some circumstances there will be no support in either target hardware or software for floating-point instructions. In these cases the `NOFP` directive can be used to ensure that no floating-point instructions or directives are allowed in the code.

`RLIST` The `RLIST` (register list) directive can be used to give a name to a set of registers to be transferred by LDM or STM.The syntax of this directive is:

label `RLIST` *list-of-registers*

If the `-CheckReglist` command-line option is selected, the registers in a register list must be supplied in increasing register order. Any failure to do this will result in a warning being produced. This can be used to help check that symbolic register names have not been misused.
list-of-registers is a comma-separated list of register names and/or ranges enclosed in braces. For example:

```
Context RLIST {r0-r6,r8,r10-r12,r15}
```

`ENTRY` The `ENTRY` directive declares its offset in its containing AREA to be the unique entry point to any program containing this AREA.

2.5.11 Thumb-specific directives: CODE 16, CODE32 and DATA

Thumb

`CODE16` Tells the assembler that subsequent instructions are to be interpreted as 16-bit (Thumb) instructions.

`CODE32` Tells the assembler that subsequent instructions are to be interpreted as 32-bit (ARM) instructions.

`DATA` Tells the assembler that the label is a “data-in-code” label (ie. it defines an area of data within a code segment). You *must* specify this directive if you are defining data in a code area.

2.6 Symbolic Capabilities

2.6.1 Setting constants: EQU, *, RN, FN, CP and CN

EQU and * Give a symbolic name to a fixed or program-relative value. The syntax is:

```
label EQU expression
label * expression
```

RN Defines register names. Registers can only be referred to by name. The names R0-R15, r0-r15, PC, pc, LR and lr, are predefined.

FN Defines the names of floating-point registers. The names F0-F7 and f0-f7 are predefined. The syntax is:

```
label RN numeric-expression
label FN numeric-expression
```

CP Gives a name to a coprocessor number, which must be within the range 0 to 15. The names p0-p15 are predefined.

CN Names a coprocessor register number; c0-c15 are predefined. The syntax is:

```
label CP numeric-expression
label CN numeric-expression
```

2.6.2 Local and global variables

While most symbols have fixed values determined during assembly, variables have values which may change as assembly proceeds.

GBL and LCL

The assembler supports both global and local variables. The syntax is:

```
directive variable-name
```

The scope of global variables extends across the entire source file while that of local variables is restricted to a particular instantiation of a macro (see ***2.9 Macros*** on page 2-26).

GBLA Declares a global arithmetic variable. Values of arithmetic variables are 32-bit unsigned integers.

GBLL Declares a global logical variable.

GBLS Declares a global string variable.

LCLA Declares and initializes a local arithmetic variable (initial state zero).

LCLL Declares and initializes a local logical variable (initial state false).

LCLS Declares and initializes a local string variable (initial state null string).

Variables must be declared before use with one of these directives.

SET

The value of a variable can be altered using the relevant one of the following three directives:

SETA	sets the value of an arithmetic variable
SETL	sets the value of a logical variable
SETS	sets the value of a string variable

The syntax of these directives is:

```
variable-name directive expression
```

where *expression* evaluates to the value being assigned to the variable named. For example:

```
VersionNumber SETA 21
VersionString SETS "Version 2.1"
Debug SETL {TRUE}
```

Note *When you set the value of a string variable, you must use quotes, as shown in the above example.*

2.6.3 Variable substitution: $

Once a variable has been declared, its name cannot be used for any other purpose, and any attempt to do so will result in an error. However, if the $ character is prefixed to the name, the variable's value will be substituted before the assembler checks the line's syntax. Logical and arithmetic variables are replaced by the result of performing a :STR: operation on them (see ***2.4.1 Unary operators*** on page 2-13), string variables are replaced by their value.

2.6.4 Built-in variables

There are several built-in variables. They are:

{PC} or .	Current value of the program location counter.
{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 32-bit program counter mode, and the value 26 if it is in 26-bit 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 it is 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

{PCSTOREOFFSET}	Is the offset between the address of the instructions:
	`STR PC,[...]`
	or
	`STM Rb,{... PC} instruction`
	with the value of PC stored out. This varies depending on the CPU and architecture specified.

2.7 Conditional Assembly: [, | and]

Sections of a source file may be assembled conditionally, only if certain conditions are true.

[or IF	marks the start of the condition
] or ENDIF	marks the end of the condition
| or ELSE	provides an else construct

The syntax is:

```
[ logical-expression
...code...
|
...code...
]
```

Note that `[`, `|` and `]` may not be the first character of a line. If `logical-expression` is true, the section will be assembled. If it is false, the second piece of code (with its beginning marked by | and the end by]) will be assembled instead. Lines of code skipped during conditional assembly will not be listed unless the assembler is switched from its default terse mode by the `-NOTERSE` command-line switch.

2.8 Repetitive Assembly: WHILE and WEND

The conditional looping statement, useful for generating repetitive tables, is provided in the assembler by the `WHILE...WEND` directives. This produces an assembly-time loop, not a runtime loop. Because the test for the `WHILE` condition is made at the top of the loop, it is possible that no code will be generated during assembly; lines are listed as for conditional assembly. The syntax is:

```
WHILE logical-expression
...code...
WEND
```

2.9 Macros

Macros are useful when a group of instructions and/or directives is frequently needed. The ARM assembler will replace the macro name with its definition. Macros may contain calls to other macros, nested up to 255 levels.

2.9.1 Defining a macro

Two directives are used to define a macro. The syntax is:

```
            MACRO
{$label}    macroname {$parameter1}{,$parameter2}{,$parameter3}..
            ...code...
            MEND
```

The `MACRO` directive must be followed by a macro prototype statement on the next line. This tells the assembler the name of the macro and its parameters. A label is optional, but is useful if the macro defines internal labels. Any number of parameters can be used; each must begin with $ to distinguish them from ordinary program symbols.

Within the macro body, $*label*, $*parameter*, etc., can be used in the same way as any other variables (see ***2.6.2 Local and global variables*** on page 2-23, and ***2.6.3 Variable substitution: $*** on page 2-24). They will be given new values each time the macro is called.

Note that the $*label* parameter is simply treated as another parameter to the macro. The macro itself describes which labels are defined where. The label does not represent the first instruction in the macro expansion. For instance, in a macro that uses several internal labels (eg. for loops), it is useful to define each internal label as the base label with a different suffix.

Sometimes a macro parameter or label needs to be appended by a value. The appended value should be separated by a dot, which the assembler will ignore once it has used it to recognize the end of the parameter and label.

For example:

```
$label.1
$label.loop
$label.$count
```

The end of the macro definition is signified by the MEND directive. There must be no unclosed WHILE/WEND loops or conditional assembly when the MEND directive is reached. Macro expansion terminates at MEND. However it can also be terminated with the MEXIT directive, which can be used in conjunction with WHILE/WEND or conditional assembly.

2.9.2 Setting default parameter values

Default values can be set for parameters by following them with an equals sign and the default value. If the default has a leading or trailing space, the whole value should appear in quotes, as shown below:

```
...{$parameter="default value"}
```

2.9.3 Macro invocation

A macro defined with a pattern such as:

```
$labxxxx $arg1,$arg2=5,$arg3
```

can be invoked as:

```
Labelxxxx val1,val2,val3
```

An omitted actual argument is given a null (empty string) value. To force use of the default value, use "|" as the actual argument.

Note *You cannot use an instruction name as a macro name, or as the first part of a macro name.*

3 Linker

This chapter introduces the ARM linker.

Linker

3.1 Introduction

The purpose of the ARM linker is to combine the contents of one or more object files (the output of a compiler or assembler) with selected parts of one or more object libraries, to produce an executable program.

3.1.1 Linker functions

The ARM linker performs the following functions:

- resolves symbolic references between object files
- extracts from object libraries the object modules needed to satisfy otherwise unsatisfied symbolic references
- sorts object fragments (AOF areas) according to their attributes and names, and consolidates similarly attributed and named fragments into contiguous chunks (see ***3.4 Area Placement and Sorting Rules*** on page 3-13)
- relocates (fully or partially) relocatable values
- generates an output image, possibly comprising several files (or a partially linked object file instead)

3.1.2 Linker input

The ARM linker, armlink, accepts as input:

- one or more separately-compiled or separately-assembled object files written in ARM Object Format (AOF) (see ***Chapter 14, ARM Object Format***)
- optionally, one or more object libraries in ARM Object Library Format (see ***Chapter 13, ARM Object Library Format***)

3.1.3 Linker output

The ARM linker can produce output in any of the following formats:

ARM Object Format	See ***Chapter 14, ARM Object Format***.
ARM Image Format	See ***Chapter 12, ARM Image Format***.
ARM Executable ELF Format	See ***Chapter 16, ELF File Format***.
ARM Shared Library Format	A read-only position-independent re-entrant shareable code segment (or shared library), written as a plain binary file, with a stub containing read-write data, entry veneers, etc., written in ARM Object Format (see ***Chapter 14, ARM Object Format*** for details).

Scatter-loading format

Enables a user to partition a program image into regions which can be positioned independently in memory. The linker generates the symbols necessary to allow a small piece of code to load the regions' memory at addresses different to their execution addresses. Scatter loading is described in the *Software Development Toolkit User Guide (ARM DUI 0040).*

ARM Overlay Format

A root segment written in ARM Image Format (AIF), with a collection of overlay segments, each written as a plain binary file. Overlays may be:

static — each segment is bound to a fixed address at link time

dynamic — each segment may be relocated during loading

Overlays are described in the *Software Development Toolkit User Guide (ARM DUI 0040).*

Plain binary format

Relocated to a fixed address (see ***Chapter 17, Other File Formats*** for details).

VLSI-Extended Intellec Hex Format

Suitable for driving the Compass integrated circuit design tools (see ***Chapter 17, Other File Formats*** for details).

3.2 Command Syntax

The format of the linker command is:

```
armlink options input-file-list
```

where:

options Means one or more command-line options. If an option takes an argument, a space must separate the argument from the keyword. Options are case-insensitive. In the remainder of this section, the permitted abbreviations recognized by armlink are shown underlined.

input-list Is one or more object files and libraries separated by spaces. Input files, libraries and linker options may also be given in a file used as an argument to the `-via` option. This is especially convenient when the input file list is long.

The input list is strictly ordered as given. For example:

```
file1 file2 –via vf1 file3 –via vf2 file4
```

yields the input file list:

```
file1 file2 vf1/1 vf1/2 .. file3 vf2/1 vf2/2 .. file4
```

where `vf1/1`, `vf1/2`, ... are the first, second..., files listed with `-via`. Each of the files in the input list must be in ARM Object Format (compiled or assembled files) or ARM Object Library Format (libraries).

3.2.1 General command-line options

`-debug`
Includes debug information in the output file. This is the default.

`-errors` *file*
Redirects the standard error stream to *file* (diagnostics will be filed there). This is especially useful under DOS, as stderr cannot be redirected using normal command-line redirection.

`-help`
Prints a screen of help text summarizing the linker's options and exit with a good return code.

`-info` *topic*
Prints information about a number of specified topics during the link process.

topic is a comma-separated list of keywords. A 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 files and library files.

`sizes` Gives a more detailed breakdown of the code and data sizes on an object by object basis.

`interwork` Lists all calls for which the ARM/Thumb interworking veneer was necessary.

`unused` Lists all unused AREAs, when used with the `-remove` option.

`-list` *`file`*

Redirects the standard output stream to *`file`*. This is especially useful in conjunction with `-map`, `-xref` and `-symbols`.

`-map`

Creates a map of the base and size of each area in the output image. This option is most useful in conjunction with the `-shl` and `-overlay` options. The map output is produced on the standard output stream (from where it can be redirected to a file using the host's stream redirection facilities or the `-list` option).

`-nodebug`

Turns off the inclusion of debug information in the output file. If objects are compiled or assembled without debugging information, the linker still includes low-level symbolic debugging data unless the `-nodebug` option is specified.

`-output` *`file`*

Names the linker's final output; this is often the name of the image file.

`-symbols` *`file`*

Lists each symbol used in the link step (including linker-generated symbols), and its value, to *`file`*. A filename of – (minus) names the standard output stream.

`-verbose`

Prints messages indicating progress of the link operation. Giving the option twice makes it even more verbose (this may be abbreviated to `-vv`).

`-via` *`file`*

Reads a further list of input filenames and linker options from file. There may be no more than 64 words on each line of a VIA file, and an option may not be split across more than one line. Conventionally, each filename and option is given on a separate line. There may be multiple VIA options, and VIA options may be nested.

`-xref`

Lists references between input areas (most useful with the `-overlay` option). The cross-reference list is produced on the standard output stream (from where it can be redirected to a file using the host's stream redirection facilities or `-list`).

3.2.2 Output format options

The following options each select a different output format (so are mutually exclusive):

`-aif`
: Generates an output image in executable ARM Image Format (AIF). This is the default if no output format option is given. The default load address for an AIF image is 0x8000 (32KB). Any other address (greater than 0x80) can be specified by using the `-Base` option (see ***3.2.4 Special command-line options*** on page 3-9). AIF is described in ***Chapter 12, ARM Image Format***.

`-aif -relocatable`
: Generates a relocatable AIF image which self-relocates to its load address when entered.

`-aif -relocatable -workspace` *n*
: Generates a relocatable AIF image which, when entered, copies itself to within *n* bytes of the top of memory and self-relocates to that address. For a description of `-Workspace` see ***3.2.4 Special command-line options*** on page 3-9).
Some fields of the AIF header and the self-relocation code generated by the linker can be customized by giving your versions in areas called `AIF_HDR` and `AIF_RELOC`, respectively, in the first object file in the input list. `AIF_HDR` must be exactly 128 bytes long (for further details see ***Chapter 12, ARM Image Format***).

`-aof`
: Generates partially-linked output in ARM Object Format (AOF), suitable for inclusion in a subsequent link step. AOF is described in ***Chapter 14, ARM Object Format***.

`-bin`
: Generates a plain binary image. The default load address for a binary image is 0. Any other address can be specified using the `-Base` option (see ***3.2.4 Special command-line options*** on page 3-9). Plain binary images are described in ***Chapter 17, Other File Formats***.

`-bin -aif`
: Generates a plain binary image preceded by an AIF header which describes it. This format is intended for use by simple program loaders and is the format of choice for them.
Such an image cannot be executed by loading it at its load address and entering it at its first word: the AIF header must first be discarded and the image must be entered at its entry point. As with a plain AIF image, the base address, which defaults to 0, can be set using the `-Base` option (see ***3.2.4 Special command-line options*** on page 3-9). Note that with `-bin -aif`, the base address is the address of the binary image, not the address of the AIF header (which is discarded). A separate base address can be given for the image's data segment using the `-data` option (see ***3.2.4 Special command-line options*** on page 3-9); otherwise, by default, data is linked immediately following code. This option directly supports images with code in ROM and data in RAM.

`-elf`
: Generates an ELF format image.

`-ihf`
: Generates a plain binary image encoded in VLSI Extended Intellec Hex Format. The output is ASCII-coded, is always big-endian, and is suitable for driving the Compass integrated circuit design tools (see ***Chapter 17, Other File Formats*** for details).

`-overlay file`
: Generates a statically-overlaid image, as described in *`file`*.
The output is a root AIF image together with a collection of plain binary overlay segments. Although the static overlay scheme is independent of the target system, parts of the overlay manager are not, and must be re-implemented for each target environment.
Overlays are described in the *Software Development Toolkit User Guide (ARM DUI 0040)*.

`-overlay` *`file`* `-relocatable`
: Generates a dynamically relocatable overlaid image, as described in *`file`*. The output is a relocatable AIF root image together with a collection of relocatable plain binary overlay segments. Although the dynamic overlay scheme is independent of the target system, parts of the overlay manager are not, and must be re-implemented for each target environment.
Overlays are described in the *Software Development Toolkit User Guide (ARM DUI 0040)*.

`-shl file` Generates a position-independent, re-entrant, read-only, shareable library, suitable for placement in ROM, together with a non re-entrant stub in ARM Object Format (in the file named by the `-output` keyword) which can be used in a subsequent client link step. A description of what is to be exported from the library is given in the file, which also contains the name of the file to hold the shareable library image.

`-shl file -reentrant`

As for `-shl`, except that a re-entrant stub is generated rather than a non re-entrant stub. A re-entrant stub is required if some other shared library is to refer to this one (by including the code of the re-entrant stub in it). Dually, a re-entrant stub demands a re-entrant client. Usually, a client application is not re-entrant (multi-threadable), so the default non re-entrant stub is useful more often.

`-split` Tells the linker to output the read-only and read-write image sections to separate output files. It may be used only in conjunction with `-bin` and `-ihf` image types, and is meaningful only if separate read-only and read-write base addresses have been specified (see ***3.2.4 Special command-line options*** on page 3-9).
This example produces a read-only file `file.ro` and a read-write file `file.rw`:

```
-o file -split -ro robase -rw rwbase
```

This example produces a read-only file `file` and a read-write file `file.dat`:

```
-o file -split -b robase -data rwbase
```

3.2.3 Scatter-loading command-line options

Scatter loading is described in detail in the *Software Development Toolkit User Guide (ARM DUI 0040)*.

The syntax for scatter loading is:

```
-scatter file
```

The linker generates a scatter-loaded image when this option is present on the linker command line.

Overlays

The options `-scatter` and `-overlay` are mutually exclusive. If a scatter-loaded application requires overlays, the scatter-load description file should be used to specify the overlays.

Ignored linker options

Several options are ignored when `-scatter` is present. These options are:

```
-ro -base
-base
-rw -base
-data
-split
```

Output file directory

When `-bin` is present on the command line, the output file specification is treated as a directory name. Each load region is placed in a separate file in that subdirectory. The filename becomes the load region name. Because of this, load region names must not contain characters unacceptable to the file system.

The following produces a directory named *xxxx* containing binary files:

```
-scatter file -bin -o xxxx
```

AIF files

Specifying `-aif` or `-bin -aif` generates an extended AIF file. This enables a scatter-loaded application to be packed into one file that is acceptable to the debugger. A modified form of AIF header is used.

When the `-scatter` option is used, `-aif` is equivalent to `-bin -aif.` A linker warning is generated if `-aif` is supplied without `-bin`.

The following produces a single file called *yyyy* containing an AIF header and the load regions:

```
-scatter file -aif -bin -o yyyy
```

3.2.4 Special command-line options

The options `-base`, `-entry`, `-data` and `-workspace` are each followed by a numerical argument. You can use a `0x` or `&` prefix to indicate a hexadecimal value, and the suffixes `K` and `M` to indicate multiplication by 1024 and 1024 x 1024, respectively.

The default base address for an AIF image is &8000 (=32K, =0x20K). The default base address for a binary image (`-bin`, `-bin -aif`, and `-ihf`) is 0.

`-case` Makes the matching of symbol names case insensitive.

`-dupok` Allows duplicate symbols (a warning is displayed) so that an area can be included more than once. The 2nd, 3rd, 4th, etc. copies of the area are not included in the image provided unused area elimination is enabled (see above).

`-entry` *entry-address*
`-entry` *offset*`+`*object*`(`*area*`)`

The objects included in an image must have a unique designated entry point. Usually, this is given by one of the input areas having been assembled from a source containing an ENTRY directive. Otherwise, the entry point must be given on the linker's command line. The entry point is the target of the entry branch from the image's AIF header. The entry point may be given as an absolute address or as an offset within an area within a particular object. For example:

```
-entry 8+startup(C$$code)
```

Note: *There must be no spaces within the argument to* `-entry`. *The letter's case is ignored when matching both object and area names. This latter form is often more convenient, and is mandatory when specifying an entry point for unused area elimination. (See* ***-remove*** *on page 3-11).*

`-first` *object(area)*
`-last` *object(area)*

These options place the area named *area* from the object named *object* first or last in the output. They are useful for forcing an area mapping low addresses to be placed first (typically the reset and interrupt vector addresses), or an area containing a checksum to be placed last.

`-match` *flags* Sets the symbol matching options and the default where 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

These options are usually set by configuring the armlink image when it is installed. The default value is 0x10 (treat pc-relative relocations as relocating code but do no default symbol matching). Do not override options accidentally when using `-match` from the command line.

`-nozeropad` Prevents zero-initialized areas from being expanded in images. This can be done at run time by initialization code. At run time, the `-nozeropad` option sets memory between `Image$$ZI$$Base` and `Image$$ZI$$Limit` to zero. The ARM C library does this by default. For plain binary files to decrease image size, `-nozeropad` is not the default. Binary images have their ZI area padded with zeros.

`-nounusedareas`

Does not remove AREAS unreachable from the AREA containing the entry point.

`-ro-base` *`base-address`*
`-base` *`base-address`*

Sets the base address for the output to *`base-address`*. This is the address at which an image may be loaded and executed without further relocation. If there are separate read-only and read-write sections, this is the base of the read-only section.

`-rw-base` *`data-base-address`*
`-data` *`data-base-address`*

Sets the base for the data (read-write) segment of the output to `data-base-address` rather than to `base-address+code-size`.

`-remove` Removes unused areas from the output. An area is used if it is either:

- the area containing the entry point, or
- referred to from a used area

`-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, from which the lower levels of code are missing, by connecting each reference to a missing function to a dummy function which does nothing. This option does not display warnings.

`-u` *`symbol`* As for `-unresolved`, but this option displays warnings.

3.3 Library Module Inclusion

An object file may contain references to external objects (functions and variables) that the linker will attempt to resolve by matching them to definitions found in other object files and libraries.

Usually, at least one library file is specified in the input list. A library is just a collection of AOF files stored in an ARM Object Library Format file. 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 (although unused areas will be eliminated from outputs of type AIF)
- a module from a library is included in the output if, and only if, an object file or an already-included library module makes a *non-weak* reference to it

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, such as references which are allowed to remain unsatisfied, and which do not cause a library member to be loaded.
2. The libraries are processed in the order that they appear in the input file list, as follows:
   a) The library is searched for members containing symbol definitions which match currently unsatisfied, non-weak references.
   b) Each such member is loaded, satisfying some unsatisfied references (including possibly *weak* ones), and maybe, creating new unsatisfied references (again, maybe including weak ones).
   c) The search is repeated until no further members are loaded.

Each library is processed in turn, so a reference from a member of a later library to a member of an earlier library cannot be satisfied. As a result, circular dependencies between libraries are forbidden.

It is an error if any *non-weak* reference remains unsatisfied at the end of a linking operation, other than one which generates partially-linked, relocatable AOF.

To forcibly include a library module, put the name(s) of the library module(s) in parentheses after the library name. Note that there should be no space between the library name and the opening parenthesis. Multiple module names must be separated by a comma. There must be no space in the list of module names.

3.4 Area Placement and Sorting Rules

Each object module in the input list, and each subsequently included library module, contains at least one area. AOF areas are the fragments of code and data manipulated by the linker.

In all output types other than AOF, the linker sorts the set of areas first by attribute, then by area name, except where overridden by a `-first` or `-last` option. The `-first` and `-last` options can be used to force particular areas to be placed first or last, regardless of their attributes, names or positions in the input list.

The read-only parts of the image are collected into one contiguous region which can be protected at runtime on systems that have memory management hardware. Page alignment between the read-only and read-write portions of the image can be forced using the area alignment attribute of AOF areas, set using the following attribute of the ARM assembler AREA directive:

`ALIGN=`*n*

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 which 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

Debugging tables are included only if the linker's `-debug` option is used. (This is the default.) A debugger is expected to retrieve the debugging tables before the image is entered. The image is free to overwrite its debugging tables once it has started executing.

Areas not ordered by attribute are ordered by AREA name. The comparison of names is lexicographical and case-sensitive, using the ASCII collation sequence for characters. Identically attributed and named areas are ordered according to their relative positions in the input list.

In some image types (AIF, for example), zero-initialized data is created at image-initialization time and does not appear in the image itself.

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.

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 which is a multiple of:

$2^{(\text{area alignment})}$

(area alignment is commonly 2, for word alignment).

3.5 Linker Predefined Symbols

There are several symbols which the linker defines independently of any of its input files. The most important of these start with the string `Image$$`. These symbols, along with all other external names containing $$, are reserved by ARM. See the *Software Development Toolkit User Guide (ARM DUI 0040)* for details of the symbols generated by the `-scatter` option.

Image-related symbols

`Image$$RO$$Base`	Address of the start of the read-only area (usually contains code).
`Image$$RO$$Limit`	Address of the byte beyond the end of the read-only area. `Image$$RO$$Limit` need not be the same as `Image$$RW$$Base`, although it often will be in simple cases of `-aif` and `-bin` output formats.
`Image$$RW$$Base`	Address of the start of the read-write area (usually contains data) `Image$$RW$$Base` is generally different from `Image$$RO$$Limit` if: • the `-data` option is used to set the image's data base (`Image$$RW$$Base`); • either of the `-shl` or `-overlay` options is used to create a shared library or overlaid image, respectively Do not rely on `Image$$RO$$Limit` being the same as `Image$$RW$$Base`.
`Image$$RW$$Limit`	Address of the byte beyond the end of the read-write area.
`Image$$ZI$$Base`	Address of the start of the zero-initialized area (zeroed at image load or startup time).
`Image$$ZI$$Limit`	Address of the byte beyond the end of the zero-initialized area.

Object/area-related symbols

`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`.

3.5.1 Notes

The read-write (data) area may contain code, as programs sometimes modify themselves (or better, generate code and execute it). Similarly, the read-only (code) area may contain read-only data, (for example string literals, floating-point constants, ANSI C const data).

These symbols can be imported and used as relocatable addresses by assembly language programs, or referred to as `extern` addresses from C (using the `-fC` compiler option which allows dollars in identifiers). Image region bases and limits are often of use to programming language runtime systems.

Other image formats (for example shared library format) have linker-defined symbolic values associated with them. These are documented in the relevant sections in this chapter.

3.6 Handling Relocation Directives

This section describes how the linker implements the relocation directives defined by ARM Object Format.

3.6.1 The subject field

A relocation directive describes the relocation of a single subject field, which may be:

- a byte
- a halfword (2 bytes)
- a word (4 bytes)
- a value derived from a suitable sequence of instructions

The relocation of a word value cannot overflow. In the other cases, overflow is detected and faulted by the linker. This is described in ***3.6.7 The relocation of instruction sequences*** on page 3-17.

3.6.2 The relocation value

A relocation directive refers either to the value of a symbol, or to 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.

3.6.3 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 and:

```
subject-field = subject-field - base-of-area-containing (subject-field)
```

which caters for a PC-relative branch to a fixed location, for example.

3.6.4 Forcing use of an inter-link-unit entry point

A second special case of PC-relative relocation applies when the relocation value is the value of a code symbol. (This is specified by `REL_B` being set in the `rel_flags` field; see ***14.1.3 AOF and the linker*** on page 14-3 for details.) It requests that the *inter*-link-unit value of the symbol be used, rather than the *intra*-link-unit value. Unless the symbol is marked with the `SYM_LEAFAT` attribute (by a compiler or via the assembler's `EXPORT` directive), the *inter*-link-unit value will be 4 bytes beyond the *intra*-link-unit value.
This directive allows the tail-call optimization to be performed on re-entrant code. For more information about tail-call continuation, see ***5.5 Function Entry*** on page 5-15.

3.6.5 Additive relocation

A plain additive relocation directive requests that the subject field be modified as follows:

```
subject-field = subject-field + relocation-value
```

3.6.6 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)
```

For example, when compiling re-entrant code, the C compiler places address constants in an *adcon* area called `sb$$adcons` based on register sb, and generates code to load them using sb-relative LDRs. At link time, separate adcon areas are merged, so sb no longer points where presumed at compile time (except for the first area in the consolidated group). The offset field of each LDR (other than those in the first area) must be modified by the offset of the base of the appropriate adcon area in the consolidated group:

```
LDR-offset = LDR-offset + base-of-my-sb$$adcons-area
             - sb$$adcons$$Base
```

3.6.7 The relocation of instruction sequences

The linker recognizes that the following instruction sequences define a relocatable value:

- a B or BL
- an LDR or STR
- 1 to 3 ADD or SUB instructions, having a common destination register and a common intermediate source register, and optionally followed by an LDR or STR with that register as base

Thumb 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 the BL instruction.

For example, the following is a relocatable instruction sequence:

```
ADD     Rb, rx, #V1
ADD     Rb, Rb, #V2
LDR     ry, [Rb, #V3]
```

with value `V = V1+V2+V3`.

The length of sequence recognized may be further restricted to 1, 2 or 3 instructions only by the relocation directive itself. For more information, see ***3.6 Handling Relocation Directives*** on page 3-16.

After relocation, the new value of V is split between the instructions as follows:

- If the original offset in the LDR or STR can be preserved, it will be preserved. This is possible if the difference between the new value and the original LDR offset can be encoded in the available number of ADD/SUB instructions. This preservation allows a sequence of ADDs and SUBs to compute a common base address for several following LDRs or STRs.

The remainder of the new value is split between the ADDs or SUBs as follows:

- If the new value is negative, it is negated, ADDs are changed to SUBs (or vice versa) and LDR/STR up is changed to LDR/STR down (or vice versa).
- Each ADD or SUB instruction in turn removes the most significant part of the (now positive) new value, which can be represented by an 8-bit constant, shifted left by an even number of bit positions which can be represented by an ARM data-processing instruction's immediate value.

If there is no following LDR or STR, and the value remaining is nonzero, the relocation has overflowed.

If there is a following LDR or STR, any value remaining is assigned to it as an immediate offset. If this value is greater than 4095, the relocation has overflowed.

In the relocation of a B or BL instruction, word offsets are converted to and from byte offsets. A B or BL is always relocated by itself, never in conjunction with any other instruction.

3.7 Automatic Inclusion of C libraries

The ARM Linker automatically searches for a C library which matches the attributes of the object files being linked.

To do this, the C libraries' filenames are annotated with letters and digits to idebtify them. The annotation has the form:

`_<apcs-variant>.<bits><bytesex>`

If the library uses all default apcs options, the annotation is just:

`.<bits><bytesex>`

3.7.1 For ARM libraries

bits is 32 or 26

bytesex is either:

- `l` little-endian
- b big-endian

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 (software stack checking is turned on if this option is not specified)

with an interworking option:

- `i` compiled for arm/thumb interworking (arm/thumb interworking is turned off if this option is not specified)

and a frame pointer option:

- n pcs uses no frame pointer (a frame pointer is used if this option is not specified)

Example

The following example is an ARM little-endian C library, using hardware floating-point instructions and with no software stack checking:

```
armlib_hc.32l
```

3.7.2 For THUMB libraries

bits is always 16

bytesex is either:

- `l` little-endian
- b big-endian

apcs-variant is the concatenation of a software stack checking option:

- s software stack checking (software stack checking is turned off if this option is not specified)

and an interworking option:

- `i` compiled for arm/thumb interworking (arm/thumb interworking is turned off if this option is not specified)

Example

The following example is a Thumb little-endian C library with software stack checking:

```
armlib_s.16l
```

Note *Not all combinations are possible; for example .16 implies no frame pointer, software floating point. Only a subset of the possible combinations are made by ARM as part of a release.*

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.

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 referencesafter 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 via the graphical configurer), or failing that, from the directory which is the value of the environment variable *ARMLIB*.

4 Rebuilding the C Library

This chapter describes how you rebuild the C libraries.

4.1 Introduction to the Runtime Libraries

Retargeting the ANSI C library requires some knowledge of ARM assembly language, and some understanding of the ARM processor and hardware being used. You need to refer to the following:

- the relevant ARM datasheet
- *section **2.3 Assembly Language Overview***
- *ARM Architecture Reference Manual (ARM DDI 0100)*

There are two runtime libraries provided to support cross-compiled C:

- the minimal embedded C library
- the ANSI C library

The libraries are supplied in:

- source form for retargeting to your ARM-based hardware
- binary form, targeted at the ARMulator (so you can immediately run and debug programs running on an emulated ARM)

Using the embedded C Library

If you intend to use the C Library in an embedded form (for example, linking with an application running from read-only memory on a target card, please refer to the *Software Development Toolkit User Guide (ARM DUI 0040)*.

4.1.1 Source files

The supplied source structure holds the following directories:

`stdh` Contains the ANSI header files (which require no change in retargeting). These files are also built into armcc.

`util` Contains the source of the `makemake` utility, written in classic C.

`semi` Contains targeting code for the semihosted C Library, which targets the debug monitor supported by:

- the ARMulator
- ARM Platform Independent Evaluation (PIE) card for the ARM60

together with SunOS-hosted make definitions and library build options. The library is called *semihosted* because many functions such as file I/O are implemented on the host computer, via the host's C library. In principle, a targeting of the library requires both a target directory and a host directory; however, where there is only one hosting, it is convenient to amalgamate the two directories.

`thumb`	Contains Thumb-specific C Library assembly code together with Thumb make definitions and library build options.
`fplib`	Contains source code for the software floating-point library.
`fpe340`	Contains object code of the floating-point emulator (for which source code is not provided).
`*.c, *.h, *.s` (top-level)	Contain target-independent source code.

The target-independent code is generally grouped into one file per section of the ANSI library (though with exceptions; stdlib is implemented partly in `alloc.c` and partly in `stdlib.c`), with use of conditional compilation or assembly to enable construction of a fine-grain library (approximately one object file per function). The ARMulator-targeted code is similarly grouped.

4.1.2 ANSI C library

The full ANSI C library contains the following:

- target-independent modules written in ANSI C; for example:
  `printf()`
- target-independent modules written in ARM assembly language: for example:
  `divide()`
  `memcpy()`
- target-dependent modules written in ANSI C; for example: default signal handlers like the `clock()` module
- target-dependent modules written in ARM assembly language

The target-independent portions of the library can be built immediately, but you need to make some modifications to the target-dependent parts to implement them. The library has a modular structure, so you do not have to retarget it in its entirety; re-target only what will be used.

The retargetable ARM C library conforms to the ANSI C library specification. Sample code is included which targets the library at the common operating environment supported by the ARMulator, ARM Evaluation boards and ARM Development boards.

The following sections provide information on how to port the ARM C library to other targets.

4.2 Constructing a Makefile

When you retarget a library, the first stage in the procedure is to construct a makefile. To do this, you use the supplied utility program `makemake`, which allows description of library variants in a host-independent manner, and permits the building of a library on a host that severely limits the number of files in a directory.

The arguments to `makemake` are

- the name of the host directory and
- if distinct, the name of the target directory

`makemake` takes as input the files `make_sun` or `make_wat` from the host directory and `sources` and `options` from the target directory. It outputs a makefile called `Makefile` in the host directory (often, the host directory and the target directory will be the same).

Input files

In order to retarget the library, at least the following files must be provided:

`makedefs` (in the host directory)

Host-dependent definitions of tools, paths, options, etc. to include in the constructed `Makefile` for the library. Use the file `makedefs` from the `semi` directory as a template.

`options` (in the target directory)

library variant selection (a number of lines, each of the form *`option_name = value`*). See ***4.4 Retargeting the Library*** on page 4-6. Use the file options from the `semi` directory as a template.

`sources` (in the target directory)

List of objects to include in the target library, and sources from which they are to be constructed. Each line (other than those controlling variant selection) has one of the forms:

- *`object_name source_name`*
- *`object_name source_name`* [*`compiler_options`*]

where *`object_name`* lacks the `.o` extension. Variant selection involves lines of the form:

```
#if expression
#elif expression
#else
#end
```

with the obvious significance. Expression primaries are *`option_name`* = *`value`* and *`option_name`* != *`value`*, and expression operators are && and || (of equal precedence). Use the file `sources` from the `semi` subdirectory as a template, modifying it as needed.

`hostsys.h` (in the target directory)

This defines the functions which must be supplied for a full retargeting of the library, and also defines certain target-dependent values required by target-independent code. Use the file `hostsys.h` from the `semi` subdirectory as a template, changing the values in it appropriately (see ***4.4 Retargeting the Library*** on page 4-6 and ***4.5 Details of Target-dependent Code*** on page 4-9).

`config.h` (in the target directory)

This contains the hardware description. The version of this file in the `semi` directory will suffice for a little-endian ARM with mixed-endian doubles; a big-endian ARM needs `BYTESEX_ODD` defined (and `BYTESEX_EVEN` not). Little-endian floating-point values are not supported by the floating-point emulator or library.

The files containing the target-specific implementation code are also provided in the target directory.

4.3 Building a Target-specific Library

When the target-dependent files have been provided, construction of a library proceeds as follows:

1. ```
   cd util
   cc -o makemake makemake.c
   ```
   As `makemake` is written portably in *classic* C it should just compile and go. The options to C compilers vary, but most support this way of making an executable program called `makemake` from the source `makemake.c`
2. ```
   cd ..
   util\makemake targetdir [hostdir]
   ```
   `hostdir` is needed only if it is different from *`targetdir`* (under UNIX, use `util/makemake` ...)
3. Edit the makefile now produced as `hostdir\Makefile`, and choose
   - big or /little endianness
   - 26 or 32 bit APCS
4. ```
   cd hostdir
   make depend
   ```
   This augments `Makefile`; as a side-effect it also makes the assembler-sourced objects.
5. ```
   make
   ```
   This makes `armlib.o`.

4.4 Retargeting the Library

The following generic variants are available as “tick box” options in the `options` file in the target directory:

fp_type	`=linked`	Includes the object module containing the floating-point emulator in the library (and linked into any image), and a small interface module to take control of the illegal instruction vector on startup, and relinquish it on closedown.
	`=module`	Floating-point emulation is provided externally (present in ROM, for example).
	`=library`	Includes the software floating-point routines in the C library. This can be used to produce a standalone image which does not require a floating-point emulator. See the *Software Development Toolkit User Guide (ARM DUI 0040)* for information on floating-point operations.
memcpy	`=small`	`memcpy`, `memmove` and `memset` are implemented by generic C code (which attempts to do as much as possible in word units); each occupies about 100 bytes.
	`=fast`	`memmove` and `memcpy` are implemented together in assembler, which attempts to do the bulk of the move 8 words at a time using LDM/STM (about 1200 bytes). `memset` is implemented similarly (about 200 bytes).
divide	`=small`	The fully-rolled implementations.
	`=unrolled`	Unsigned and signed divide are unrolled 8 times for greater speed, but use more code. Complete unrolling of divide is possible, but should be done with care as the significant size increase might give decreased rather than increased performance on a cached ARM. Whichever variant is selected, it includes fast unsigned and signed divide by 10.
stack	`=contiguous` `=chunked`	For details, see ***4.4.2 Address space model*** on page 4-7.
stdfile_redirection		
	`=on`	`_main` extracts UNIX-style stdstream connection directives from the image’s argument string (<, >, >>, >&, 1>&2).
backtrace	`=on`	The default signal handler ends by producing a call-stack traceback to `stderr`. Use of this variant is not encouraged, since it increases the proportion of the library that is linked into all images, while providing functionality better obtained from a separate debugger.

4.4.1 Basic choices

After the tick box choices have been made, you need to make basic choices about the address-space model and the I/O model the library will follow.

4.4.2 Address space model

Two address space models are supported:

- contiguous stack
- chunked stack

Contiguous stack

Choosing `stack` = `contiguous` gives:

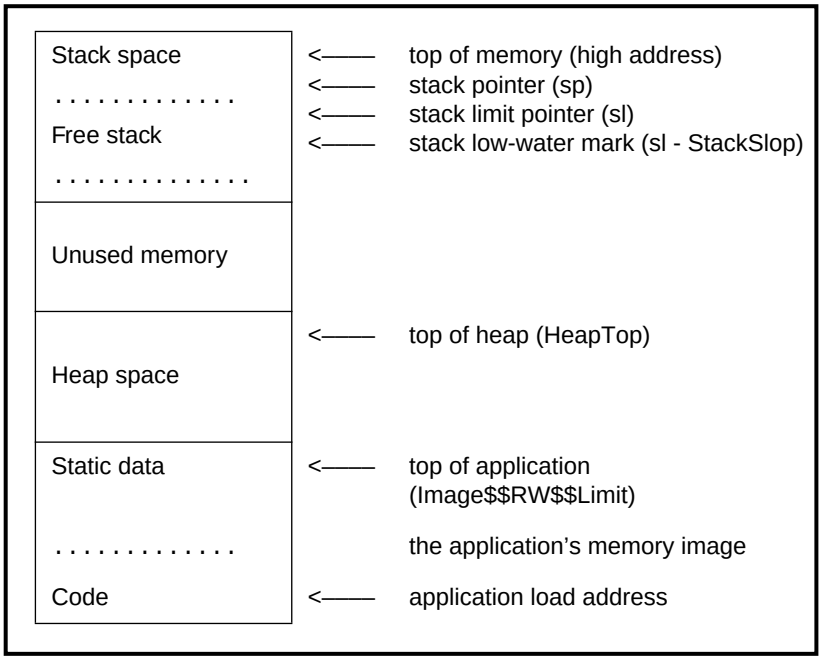


Figure 4-1: Chunked stack

Chunked stack

Choosing `stack = chunked` gives:

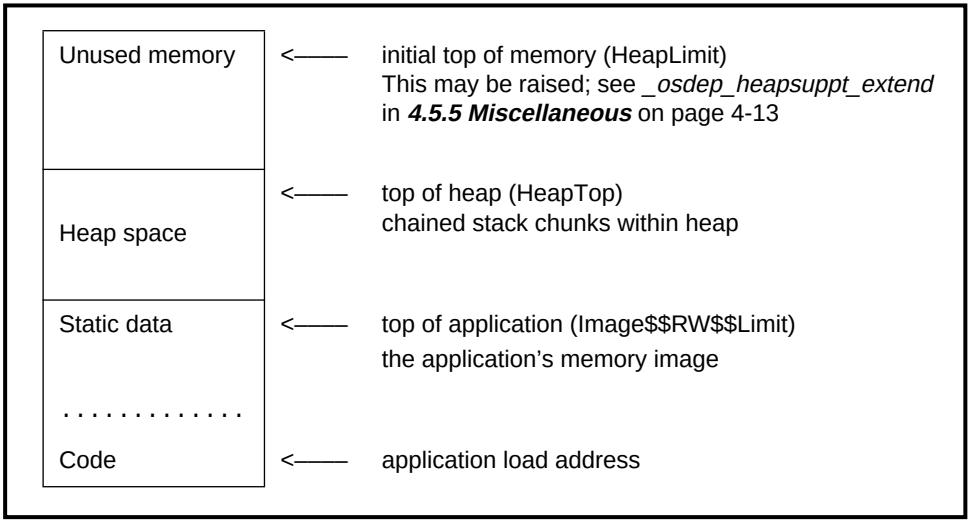


Figure 4-2: Contiguous stack

A third variant, like the first, but with the stack outside the heap and not under the application's control, can easily be synthesized. This may be a more appropriate variant if there is a skeletal operating system which implements an address-mapped stack segment.

4.4.3 I/O model

The library, as supplied, only conveniently handles byte-stream files. This does not mean that other file types cannot be handled in the target-dependent I/O support level, but that such support may be complicated; block stream files, for example, are simple to support in the absence of user-supplied buffers.

4.5 Details of Target-dependent Code

4.5.1 ANSI library functions

The following ANSI standard functions have an implementation that fully depends on the target operating system. No functions are used internally by the library (so only clients which directly call the functions will fail, if any functions are not implemented).

`clock_t clock(void)`

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, `time.h` must be adjusted to define appropriate values for `CLK_TCK` and `CLOCKS_PER_SEC`.

`void _clock_init(void)` (declared weak)

where:

`clock_init()` (if provided) is called from the library's initialization code.

`clock()` needs initializing if a read-only timer is all it has to work with.

```
time_t time(time_t *timer)
int remove(const char *pathname)
int rename(const char *old, const char *new)
int system(const char *string)
char *getenv(const char *name)
void getenv_init(void) (declared weak)
```

`getenv_init()` is called from the library's initialization code if you provide an implementation of it.

4.5.2 I/O support

If any I/O function is to be used, `hostsys.h` must define the type `FILEHANDLE`, the values of which identify an open file to the host system. There must be at least one distinguished value of this type, defined by the macro `NONHANDLE`, 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 OS 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).

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

The function `_sys_open()` is needed by `fopen()` and `freopen()`, which in turn are required if any I/O function is to be used. `openmode` is a bitmap, whose bits mostly correspond directly to the ANSI mode specification: for details, see `hostsys.h` in ***4.1.1 Source files*** on page 4-2. (Target-dependent extensions are possible, in which case `freopen()` must be extended too.)

`int _sys_iserror(int status)`

A `_sys_iserror()` function, or a `_sys_iserror()` macro, is required if any of the following int-returning functions is provided to determine whether the return value indicates an error.

`int _sys_close(FILEHANDLE fh)`

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

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

This function must be defined if any output function or `sprintf` variant is to be used. The `mode` argument is a bitmap describing the state of the FILE connected to `fh`. See the `_IOxxx` constants in `ioguts.h` for the 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 (ie. non-0 denotes a failure of some sort), or an error indicator.

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

The function `_sys_read()` must be defined if any input function or `sscanf` 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 (ie. `len` - `result` were read), or
- an error indication, or
- 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, or

  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.

`int _sys_ensure(FILEHANDLE fh)`

This function is only required if you define `HOSTOS_NEEDSENSURE` (see above). 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.

`long _sys_flen(FILEHANDLE fh)`

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. Note that it is possible to 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.

`void _ttywrch(int ch)`

This function must be defined. It writes a character, notionally to the console. 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, in the default SIGSTK handler).

`int _sys_istty(FILE *)`

This function returns non-zero if the argument file is connected to a terminal. It 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.

`void _sys_tmpnam(char *name, int fileno);`

This function returns the name for temporary file number `fileno` in the buffer `name`. It must be defined if `tmpnam()` or `tmpfil()` are to be used.

4.5.3 Floating-point support

```
int __fp_initialise(void)
void __fp_finalise(void)
```

This function returns 1 if floating-point instructions are available, otherwise 0. If the variant `fp_type == module` is selected, these two functions must be supplied (though they need not do anything).

`bool __fp_address_in_module(void *)`

This function must be provided if the variant `fp_type == module` is selected and the supplied abort handlers are used. It should return 1 if the argument address falls within the code of the fp emulator (to allow the abort handler to describe what is really an abort on a floating-point load or store as such, rather than somewhere within the emulator's code).

4.5.4 Kernel

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.

The ARMulator version of this file (`kernel.s` in the `semi` directory) can be used as a prototype.

The following are the main interfaces to the kernel:

`__main()`
: This provides the entry point to the application, and is called after low-level library initialisation. (The required initialization depends on the target environment: this may include heap, stack, and fp support, calling various `osdep_xxx_init()` functions if they exist). `__rt_lib_init` must be called to initialize the body of the library.

```
void __rt_exit(int);
```
: This mandatory function finalizes the library (including calling `atexit()` handlers), then returns to the operating system with its argument as a completion code.

```
char *__rt_command_string(void);
```
: This mandatory function returns the address of (maybe a copy of) the string used to invoke the program.

```
void __rt_trap(__rt_error *, __rt_registers *);
```
: This mandatory function handles a fault (for example, the processor detected a trap, and enabled fp exception). 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.

```
unsigned __rt_alloc(unsigned minwords, void **block);
```
: This function is the low-level memory allocator underlying `malloc()`. The `malloc()` function allocates only memory between HeapBase and HeapTop; a call to `__rt_alloc` attempts to move HeapTop: compare UNIX `sbrk()`. `__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 HeapLimit. 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 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`.

`void *(*__rt_malloc)(size_t)`

This is a function pointer, which the kernel should initialize to some primitive memory allocation function. The library itself contains no calls to `malloc()`, other than those from functions of the malloc family, such as `calloc()`. Instead the function pointed to by `__rt_malloc` is called. `__rt_malloc` is set to `malloc` during initialization (if `malloc` is linked into the image). The use of `__rt_malloc` ensures that allocations succeed, if they are made before `malloc` is initialized, and prevents `malloc` from being necessarily linked into an image, even when unused.

`extern void (*__rt_free)(void *)`

This is a function pointer, which the kernel should initialize to some primitive memory-freeing function (see `__rt_malloc` above).

4.5.5 Miscellaneous

`void __osdep_traphandlers_init(void)`

This arranges to catch processor aborts (and passes them to `__rt_trap`).

`void __osdep_traphandlers_finalise(void)`

This removes the processor abort handlers installed by ..._`init()`.

`void __osdep_heapsupport_init(HeapDescriptor *)`

This function must be provided, but may be null.

`void __osdep_heapsupport_finalise(void)`

This function must be provided, but may be null.

`{ int, void *} __osdep_heapsupport_extend(int size, HeapDescriptor *)`

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 a null version just returning 0 will suffice if heap extension is not needed.

`char *_hostos_error_string(int no, char *buf);`

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.

`char *_hostos_signal_string(int no)`

This function is called to return a string describing a signal whose number is outside the set `SIGxxx` defined in `signal.h`.

5 ARM Procedure Call Standard

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

5.1 Introduction

The *ARM Procedure Call Standard* (*APCS*) is a set of rules which regulates and facilitates calls between separately compiled or assembled program fragments.

The APCS defines:

- constraints on the use of registers
- stack conventions
- the format of a stack-based data structure, used by stack tracing programs to reconstruct a sequence of outstanding calls
- passing of machine-level arguments, and the return of machine-level results at externally visible function/procedure calls
- support for the ARM shared library mechanism; a standard way for shared (re-entrant) code to address the static data of its clients

Since the ARM CPU is used in a wide variety of systems, the APCS is not a single standard but a consistent family of standards. (See section ***5.3 APCS Variants*** on page 5-11 for details of the variants in the family.) As you implement runtime systems, operating systems, embedded control monitors, etc., you must choose the variant(s) most appropriate to your requirements.

There is no binary compatibility between program fragments which conform to different members of the APCS family. Developers who are concerned with long-term binary compatibility must choose their options carefully.

In the following, the term *function* is used to mean function, procedure or subroutine.

5.1.1 Design criteria

The APCS is a compromise between *fastest*, *smallest* and *easiest to use*; most importantly:

- function calls should be fast, and it should be easy for compilers to optimize function entry sequences
- the function call sequence should be as compact as possible
- extensible stacks and multiple stacks should be accommodated
- the standard should encourage the production of re-entrant code, with writable data separated from code
- the standard should be simple enough to be used by assembly language programmers, and should support simple approaches to link editing, debugging and runtime error diagnosis

Overall, compact code and a clear definition rank highest, with simplicity and ease of use ahead of performance in matters of fine detail where impact on performance is small.

5.2 Defining the APCS

This section defines the ARM Procedure Call Standard. Extra descriptions of features of the standard are also included; these do not form part of the standard, but are given to aid clarity.

Program fragments

A program fragment which conforms to the APCS while making a call to an external function (one which is visible between compilation units) is said to be *conforming*. A program which 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 code merely 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.

5.2.1 Register names

The ARM has:

- 15 visible general registers
- a program counter register
- eight floating-point registers.

In non-user machine modes, some general registers are shadowed. In all modes, the availability of the floating-point instruction set depends on the processor model, hardware and operating system.

In the context of the APCS, the ARM registers have the names and functions described in ***Table 5-1: ACPS registers*** on page 5-4.

5.2.2 General registers

The 16 integer registers are divided into 3 sets:

- four argument registers which can also be used as scratch registers or as caller-saved register variables
- five callee-saved registers, conventionally used as register variables
- seven registers which have a dedicated role, at least some of the time, in at least one variant of APCS-3 (see ***5.3 APCS Variants*** on page 5-11)

The five frame registers fp, ip, 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 the/a local code generator temporary register.)

There are dedicated roles for sb 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.

Number	APCS name	APCS role
r0	a1	argument 1 / integer result / scratch register
r1	a2	argument 2 / scratch register
r2	a3	argument 3 / scratch register
r3	a4	argument 4 / scratch register
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	frame pointer
r12	ip	scratch register / new-sb in inter-link-unit calls
r13	sp	lower end of current stack frame
r14	lr	link address / scratch register
r15	pc	program counter

Table 5-1: ACPS registers

5.2.3 Floating-point registers

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.

These 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 ***5.2.8 Data representation and argument passing*** on page 5-9 and ***5.3 APCS Variants*** on page 5-11
- registers f4 through f7, the *variable* registers, preserved by callees

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

Table 5-2: Floating-point registers

5.2.4 The stack

The stack is a singly-linked list of *activation records*, linked through a *stack backtrace data structure* (see ***5.2.5 The stack backtrace data structure*** on page 5-6), stored at the high-address end of each activation record.

- The stack must be readable and writable by the executing program.
- 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 may be multiple stack chunks, and there are no constraints on the ordering of these chunks in the address space.

Stack chunk limit

Associated with sp is a possibly implicit stack chunk limit, below which sp must not be decremented. (See ***5.3 APCS Variants*** on page 5-11.)

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.

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* in all other cases.

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 merely 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
- identify the same chunk as sp points into at all times

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 via an intermediate function invocation, specially fabricated when the stack was extended.)

5.2.5 The stack backtrace data structure

The value in fp must be zero or must point to a list of stack backtrace data structures which partially 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:

```
save code pointer       [fp]        <-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
```

5.2.6 Function invocations and backtrace structures

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 which 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.)

Value	Restored to
`return link value`	pc
`return sp value`	sp
`return fp value`	fp

Table 5-3: Function exit

Variant	Save code pointer
32-bit PC	Saved using the instruction: `STM{...PC}` at the start of the sequence of instructions that created the stack backtrace data structure.
26-bit PC	Saved using the instruction: `STM{...PC}` at the start of the sequence of instructions that created the stack backtrace data structure.

Table 5-4: APCS variants

5.2.7 Control arrival

At the instant when control arrives at the target function:

- pc contains the address of an entry point to the target function (re-entrant functions may have two entry points).
- lr contains the value to restore to pc on exit from the function (the `return link value`. See ***5.2.5 The stack backtrace data structure*** on page 5-6).
  (In 26-bit variants of the APCS, lr contains the PC + PSR value to restore to pc on exit from the function: see ***5.3 APCS Variants*** on page 5-11.)
- 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 ***5.2.4 The stack*** on page 5-5.
- fp contains 0 or points to the most recently created stack backtrace structure: see ***5.2.5 The stack backtrace data structure*** on page 5-6.
- The space between sp and the stack chunk limit is readable and writable memory which the called function can use as temporary workspace and overwrite with any values before the function returns: see ***5.2.4 The stack*** on page 5-5.
- Arguments are marshalled as described in ***5.2.8 Data representation and argument passing*** on page 5-9.

A re-entrant target function (see ***5.3 APCS Variants*** on page 5-11) 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

(Sometimes the two entry points are at the same address. They are normally separated by a single instruction.)

On arrival at the intra-link-unit entry point, sb *must* identify the static data of the link unit which contains both the caller and the callee.

On arrival at the inter-link-unit entry point, ip must identify the static data of the link unit containing the target function, or the target function *must* make neither direct nor indirect use of static data.

(In practice this usually means the callee must be a leaf function making 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 which will be returned to if the tail continuation is converted to a return.

If code is not required to be re-entrant or shareable, sb may be used as v6, an additional variable register. (See ***Table 5-1: ACPS registers*** on page 5-4.)

5.2.8 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:

- a word-sized, integer value, or
- 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 and so forth, used by ARM-targeted compilers for C, Pascal, Fortran-77, etc., nor does it prescribe the order in which language-level arguments are mapped into their machine-level representations. In other words, 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 ***5.4 C Language Calling Conventions*** on page 5-13.

At the instant control arrives at the target function, the argument list must be allocated as follows:

- in APCS, variants which support the passing of floating-point arguments in floating-point registers (see ***5.3 APCS Variants*** on page 5-11), 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
- 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.

5.2.9 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 ***5.4 C Language Calling Conventions*** on page 5-13).
- If the function returns a simple floating-point value, the value *must* be in f0 for hardfp. For softfp, the results are 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.

In 32-bit ARM modes, the caller's PSR flags are not preserved across a function call. In 26-bit ARM modes, the caller's PSR flags are naturally reinstated when the return link pointer is placed in pc.

Note that the N, Z, C and V flags from lr at the instant of entry must be reinstated; it is not sufficient merely to preserve the PSR across the call.

Consider a function `ProcA` which tail continues to `ProcB` as follows:

```
CMPS   a1, #0
MOVLT  a2, #255
MOVGE  a2, #0
B      ProcB
```

If `ProcB` just preserves the flags it sees on entry, rather than restoring flags from lr, the wrong flags may be set when `ProcB` returns direct to `ProcA`'s caller. See ***5.3 APCS Variants*** on page 5-11.

5.3 APCS Variants

There are 16 APCS variants, derived from four independent choices (2 x 2 x 2 x 2). In each case, code conforming to one variant is not compatible with code conforming to another. The only true user-level choice is between re-entrant versus non-re-entrant variants:

- 32-bit PC vs 26-bit PC. This is fixed by your ARM CPU.
- Implicit vs explicit stack-limit checking. This is fixed by a combination of memory-management hardware and operating system software. Use implicit stack-limit checking if your ARM-based environment supports it; otherwise use explicit stack-limit checking.
- Passing floating-point arguments. This supports efficient argument passing where:
  a) the floating-point instruction set is emulated by software, and floating-point operations are dynamically very rare, or
  b) the floating-point instruction set is supported by hardware, or floating-point operations are dynamically common

As these alternatives are compatible, each may be used where appropriate.

5.3.1 32-bit PC vs 26-bit PC

Older ARM CPUs and the 26-bit compatibility mode of newer CPUs use a 24-bit, word-address program counter, by packing:

- four status flags (NZCV) and two interrupt-enable flags (IF) into the top six bits of r15
- two mode bits (m0, m1) into the least-significant bits of r15

Thus, r15 implements a combined PC + PSR.

Newer ARM CPUs use a 32-bit program counter (in r15) and a separate PSR.

In 26-bit CPU modes, the PC + PSR is written to r14 by an ARM branch with link instruction, so it is natural for the APCS to require the reinstatement of the caller's PSR at function exit (a caller's PSR is preserved across a function call).

In 32-bit CPU modes, this reinstatement is unacceptably expensive in comparison to the gain from it, so the APCS does not require it, and a caller's PSR flags may be corrupted by a function call.

5.3.2 Implicit vs explicit stack-limit checking

ARM-based systems vary widely in the sophistication of their memory management hardware. Some can easily support multiple, auto-extending stacks, while others have no memory management hardware at all. However, the majority of ARM-based systems require software stack-limit checking.

Safe programming practices demand that stack overflow be detected. The APCS defines conventions for software stack-limit checking sufficient to support efficiently most requirements (including those of multiple threads and chunked stacks).

5.3.3 Floating-point arguments in floating-point registers

Many ARM-based systems have made no use of the floating-point instruction set, or have used a software emulation of floating-point instructions.

Systems using a slow software emulation, which makes little use of floating-point, have a small disadvantage in passing floating-point arguments in floating-point registers; all variadic functions (such as `printf`) become slower, while only function calls which actually take floating-point arguments become faster.

Note *If your system has no floating-point hardware and is expected to make little use of floating-point, it is better not to pass floating-point arguments in floating-point registers.*

5.3.4 Re-entrant vs non-re-entrant code

The re-entrant variant of the APCS supports the generation of code that is free of relocation directives. This is position-independent code which addresses all data indirectly via a static base register. Such code is ideal for placement in ROM and can be shared between several client processes.

In general, code to be placed in ROM or loaded into a shared library is expected to be re-entrant, while applications are not expected to be re-entrant.

See also ***5.4 C Language Calling Conventions*** on page 5-13.

5.4 C Language Calling Conventions

5.4.1 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. (See the *Software Development Toolkit User Guide (ARM DUI 0040)* for more information.)

The C compiler widens arguments of type float to type double to support interworking between ANSI C and classic C.

Char, short, pointer and other integral values occupy one word in an argument list. Char and short values are widened by the C compiler during argument marshalling.

On the ARM, characters are naturally unsigned. In PCC mode (`-pcc` option), the C compiler treats a plain char as signed, widening its value appropriately when used as an argument. (Classic C lacks the signed char type, so plain chars are considered signed; ANSI C has signed, unsigned and plain chars.)

A structured value occupies an integral number of integer words (even when it contains only floating-point values).

5.4.2 Argument list marshalling

Argument values are marshalled in the order written in the source program.

If passing floating-point (FP) arguments in FP registers, the first four FP 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 (so that arguments later in the argument list have higher addresses than those earlier in the argument list). As a consequence, an FP value can be passed in integer registers, or even split between an integer register and the stack.

This follows from the need to support variadic functions (functions that have a variable number of arguments, such as `printf`, `scanf`, etc.). Alternatives which avoid passing FP values in integer registers require the caller to know that a variadic function is being called, and use different argument marshalling conventions for variadic and non-variadic functions.

5.4.3 Non-simple value return

A non-simple type is any non floating-point type of size 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 sub-fields 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;} and 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, ...);
```

5.5 Function Entry

A complete discussion of function entry is complex; this section covers a few of the most important issues and special cases.

The important issues for function entry are:

- establishing the static base (if the function is to be re-entrant)
- creating the stack backtrace data structure (if needed)
- saving the floating-point variable registers (if required)
- checking for stack overflow (if the stack chunk limit is explicit)

Leaf functions

A function is termed *leaf* if its body contains no function calls.

A leaf function which makes no use of static data need not establish a static base.

Tail calls or tail continuations

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.

V-registers

A function does not need to create a stack backtrace structure if it uses no v-registers and:

- it is a leaf function, or
- all the function calls it makes from its body are tail calls

Such functions are termed *frameless.*

5.5.1 Establishing the static base

The ARM shared library mechanism supports both:

- direct linking together of functions into a *link unit*
- indirect linking of functions with the *stubs* of other link units

Thus a re-entrant function can be entered directly via a call from the same link unit (an *intra-link-unit call*), or indirectly via a function pointer or direct call from another link unit (an *inter-link-unit call*).

The general scheme for establishing the static base in re-entrant 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 which does not have to be re-entrant does not need to use a static base. Code which is re-entrant is marked as such, allowing the linker to create the inter-link-unit veneers needed between independent re-entrant link units, and between re-entrant and non-re-entrant code.

5.5.2 Creating the stack backtrace structure

For non re-entrant, nonvariadic functions, the 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 a 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 re-entrant 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 re-entrant 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, re-entrant 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 re-entrant variadic functions, use:

```
MOV    ip, sp                          ; save current sp, ready to
                                       ; save as old sp
STMFD  sp!, {a1-a4}                    ; push arguments on stack
SFMFD  f0, 4, [sp]!                    ; push FP arguments on
STMFD  sp!, {v1-v6, fp, ip, lr, pc}    ; 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).

If floating-point arguments are not being passed in floating-point registers, there is no need for SFMFD. SFM is not supported by the Issue 1 floating-point instruction set and must be simulated by four STFE instructions. See section ***5.5.3 Saving and restoring floating-point registers*** on page 5-17.

For re-entrant variadic functions, the requirements are more complex.

5.5.3 Saving and restoring floating-point registers

The Issue 2 floating-point 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 (eg. {f4–f7}, {f6, f7, f0}, {f7, f0}, {f1})

In Issue 1 floating-point instruction set compatibility modes, SFM and LFM must be simulated using sequences of STFEs and LDFEs.

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.
```

On function exit, sp-relative addressing may be unavailable if the stack has been discontiguously extended.

5.5.4 Checking for stack limit violations

In some environments, stack overflow detection will be implicit; an off-stack reference will cause 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()`.

In Modula-2, Pascal and other languages, arrays may be created on block entry or passed as *open array arguments,* the size of which is unknown until runtime. These are located in the callee's stack frame, and so impact stack-limit checking. In practice this is not a problem—see ***5.5.7 Stack limit checking (vari-sized frames for Pascal-like languages)*** on page 5-19.

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 (`__rt_stkovf_split_small` or `__rt_stkovf_split_big`).

Any environment that supports explicit stack-limit checking must provide functions which 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

5.5.5 Stack limit checking (small, fixed frames)

For frames of 256 bytes or less the limit check is as follows:

```
create stack backtrace structure

CMPS   sp, sl
BLLT   |__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.

5.5.6 Stack limit checking (large, fixed frames)

For frames bigger than 256 bytes, the limit check proceeds as follows:

```
SUB    ip, sp, #FrameSizeBound       ; can be done in 1 instr
CMPS   ip, sl
BLLT   |__rt_stkovf_split_big|
SUB    sp, sp, #InitFrameSize        ; may take more than 1 instr
```

Note *Functions containing nested blocks may use different amounts of stack at different instants during their execution.*

`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.

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.

5.5.7 Stack limit checking (vari-sized frames for Pascal-like languages)

The handling of frames whose size is unknown at compile time is identical to the handling of large frames, with the following exceptions:

- the computation of the proposed new stack pointer is more complicated, involving arguments to the function itself
- the addressing of vari-sized objects is more complicated than the addressing of fixed size objects
- vari-sized objects have to be initialized by the called function

Stack layout

The general scheme for stack layout in this case is shown in ***Figure 5-1: Stack layout***:

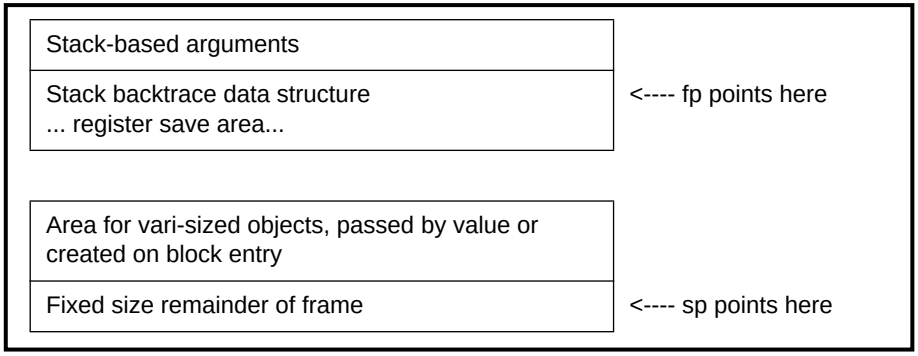


Figure 5-1: Stack layout

Objects notionally passed by value are actually passed by reference and copied by the callee.

The callee addresses the copied objects via pointers located in the fixed size part of the stack frame, immediately above sp. These can be addressed relative to sp. The original arguments are all addressable relative to fp.

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.

If a nested block extends the stack by an amount which cannot be known until runtime, the block entry must include a stack limit check.

5.5.8 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. (Fortran may violate this rule by virtue of multiple entries.)

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 ***5.5.3 Saving and restoring floating-point registers*** on page 5-17.

To exit from functions that use no stack and save no floating-point registers, use:

```
MOV    pc, lr
```

To exit from other functions which save no floating-point registers, use a pre-decrement multiple load (`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.

26-bit compatibility

Note *These instructions must not be used in 32-bit mode.*

To exit from functions that use no stack and save no floating-point registers, note that 26-bit compatibility demands the following instruction to reinstate the caller's PSR flags:

```
MOVS  pc, lr
```

To exit from other functions which save no floating-point registers, use a pre-decrement multiple load (LDMEA) to reinstate the caller's PSR flags:

```
LDMEA fp, {regs}^
```

5.5.9 Some examples

This section highlights some of the optimizations that are particularly relevant to the ARM and to this standard.

In order to make effective use of the APCS, compilers must compile code one procedure at a time; compiling one line at a time is insufficient.

In the case of leaf functions, much of the standard entry sequence can be omitted.
In very small functions, such as those that frequently occur implementing data abstractions, the function-call overhead can be tiny.

Consider:

```
typedef struct {...; int a; ...} foo;
int foo_get_a(foo* f) {return(f-a);}
```

The function foo_get_a can compile to just:

```
LDR  a1, [a1, #aOffset]
MOV   pc, lr                          ; MOVS in 26-bit modes
```

In functions with a conditional as the top-level statement, in which one or other arm of the conditional is *leaf* (calls no functions), the formation of a stack frame can be delayed.

For example, the C function:

```
int get(Stream *s)
{
    if (s->cnt > 0)
    {--s->cnt;
        return *(s->p++);
    }
    else
    {
        ...
    }
}
```

could be compiled (non-re-entrantly) into:

```
get MOV    a3, a1
; if (s->cnt > 0)
    LDR    a2, [a3, #cntOffset]
    CMPS   a2, #0
; try the fast case,frameless and heavily conditionalized
    SUBGT  a2, a2, #1
    STRGT  a2, [a3, #cntOffset]
    LDRGT  a2, [a3, #pOffset]
    LDRBGT a1, [a2], #1
    STRGT  a2, [a3, #pOffset]
    MOVGT  pc, lr
; else, form a stack frame and handle the rest as normal code.
    MOV    ip, sp
    STMDB  sp!, {v1-v3, fp, ip, lr, pc}
    CMP    sp, sl
    BLLT   |__rt_stkovf_split_small|
    ...
    LDMEA  fp, {v1-v3, fp, sp, pc}
```

This is only worthwhile if the test can be compiled using any spare of a1-a4 and ip as scratch registers. This technique can significantly accelerate certain speed-critical functions, such as read and write character.

Finally, it is often worth applying the tail call optimization, especially to procedures which need to save no registers.

For example:

```
extern void *malloc(size_t n)
{
    return primitive_alloc(NOTGCABLEBIT, BYTESTOWORDS(n));
}
```

is compiled (non-re-entrantly) by the C compiler into:

```
malloc
    ADD    a1, a1, #3               ; 1S
    MOV    a2, a1, LSR #2           ; 1S - BITESTOWORDS(n)
    MOV    a1, #1073741824          ; 1S - NOTGCABLEBIT
    B      primitive_alloc          ; 1N+2S = 4S
```

In this case, the optimization avoids saving and restoring the call-frame registers and saves five instructions (and many cycles; 17 S cycles on an uncached ARM with N=2S).

5.6 The APCS in Non-user ARM Modes

There are some consequences of the ARM's architecture which need to be understood by implementors of code intended to run in the ARM's SVC and IRQ modes.

- An IRQ corrupts r14_irq, so IRQ-mode code must run with IRQs off until r14_irq has been saved. A general solution to this problem is to enter and exit IRQ handlers written in high-level languages via hand-crafted wrappers, which:

on entry	save r14_irq, change mode to SVC, and enable IRQs
on exit	restore the saved r14_irq, IRQ mode and the IRQ-enable state.

  Thus the handlers themselves run in SVC mode, avoiding the problem in compiled code.
- SWIs corrupt r14_svc, so care has to be taken when calling SWIs in SVC mode.

  In high-level languages, SWIs are usually called out of line, so you only need to save and restore r14 in the calling veneer around the SWI. If a compiler can generate inline SWIs, it should also generate code to save and restore r14 inline around the SW using the `-fz` option, unless you know that the code will not be executed in SVC mode.

5.6.1 Aborts and pre-ARM6-based ARMs

With pre-ARM6-based ARMs (ARM2, ARM3), aborts corrupt r14_svc. This means that care has to be taken when causing aborts in SVC mode.

An abort in SVC mode may be symptomatic of a fatal error, or it may be caused by page faulting in SVC mode. Page faulting can occur because an instruction needs to be fetched from a missing page (causing a prefetch abort), or as a result of an attempted data access to a missing page. The latter may occur even if the SVC-mode code is not itself paged (consider an unpaged kernel accessing a paged user-space).

Data aborts

A data abort is recoverable provided r14 contains nothing of value at the instant of the abort. This can be ensured by:

- saving R14 on entry to every function and restoring it on exit
- not using R14 as a temporary register in any function
- avoiding page faults (stack faults) in function entry sequences

You can use a software stack-limit check to avoid data-aborts early in function entry sequences.

Prefetch aborts

A prefetch abort is harder to recover from, and an aborting `BL` instruction cannot be recovered, so special action has to be taken to protect page faulting function calls.

In code compiled from C, r14 is saved in the second or third instruction of an entry sequence. Aligning all functions at addresses which are 0 or 4 modulo 16 ensures the critical part of the entry sequence cannot prefetch-abort. A compiler can do this by padding code sections to a multiple of 16 bytes, and being careful about the alignment of functions within code sections.

A possible way to protect BL instructions from prefetch-aborts is to precede each BL by:

```
MOV    ip, pc
```

If the BL faults, the prefetch abort handler can safely overwrite r14 with ip before resuming execution at the target of the BL. If the prefetch abort is not caused by a BL, this action is harmless, as r14 has been corrupted anyway (and, by design, contained nothing of value when a prefetch abort could occur).

6 Thumb Procedure Call Standard

This chapter describes the Thumb procedure call standard.

Thumb Procedure Call Standard

6.1 Introduction

The *Thumb Procedure Call Standard* (*TPCS*) is a set of rules that govern inter-calling between functions written in the Thumb subset of ARM.

The TPCS is heavily based on the APCS. If you are unfamiliar with the APCS and its terminology, you will find it helpful to read ***Chapter 5, ARM Procedure Call Standard*** before continuing with this chapter.

In essence, the TPCS is a cut-down version of the APCS. This reduction in versatility reflects the different ways in which ARM and Thumb code is used, and also reflects the reduced nature of the Thumb instruction set, which makes implementing the full APCS inappropriate.

Specifically, the TPCS does not allow:

- Disjoint stack extension (stack chunks).
  Under TPCS, the stack must be contiguous. However, this does not necessarily prohibit the use of co-routines. There are various methods for implementing co-routines on a contiguous stack. For example, many C++ run-time library co-routines are implemented in this way.
- Calling the same entry point with different sets of static data (multiple instantiation).
  Multiple instantiation can still 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.
- Direct floating-point support.
  Thumb cannot have access to floating-point (FP) instructions without switching to ARM mode. Floating-point is supported indirectly by defining how FP values are passed to and returned from Thumb functions in the Thumb registers.

6.2 Register Names

The Thumb register subset has:

- eight visible general-purpose registers (r0-r7)
- a stack pointer (SP)
- a link register (LR)
- a program counter (PC)

In addition, the Thumb subset can access all of the ARM registers singly via a set of special instructions. See the *ARM Architecture Reference Manual* for details.

In the context of the TPCS, each Thumb register has a special name and function:

Register	TPCS Name	TPCS Role
0	a1	argument 1 / scratch register / FP result / integer result
1	a2	argument 2 / scratch register / FP result
2	a3	argument 3 / scratch register / FP result
3	a4	argument 4 / scratch register
4	v1	register variable
5	v2	register variable
6	v3	register variable
7	v4/wr	register variable/work register in function entry/exit
8	(v5)	(ARM v5 register; no defined role in Thumb)
9	(v6)	(ARM v6 register; no defined role in Thumb)
10	sl	stack limit
11	fp	frame pointer
12	(ip)	(ARM ip register - no defined role in Thumb)
13	sp	stack pointer (full descending)
14	lr	link address
15	pc	program counter

Table 6-1: TPCS registers

6.3 The Stack

The stack contains a series of activation records allocated in descending order. These activation records may be linked through a stack backtrace data structure.

Note *There is no obligation for code under TPCS to create a stack backtrace data structure. This facility is principally included for use by code compiled for debugging purposes.*

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: it 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.

6.3.1 Implicit vs explicit stack limit checking

Stack limit checking may be implicit or explicit. This is fixed by a combination of memory-management hardware and system software.

Safe programming practices demand the detection of stack overflow. Although the majority of Thumb-based systems are expected to have hardware stack limit checking, the TPCS defines conventions for software stack-limit checking sufficient to support most requirements.

6.4 Control Arrival and Return

6.4.1 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 ***5.2.5 The stack backtrace data structure*** on page 5-6).
- sp points at or above the current stack limit. If the limit is explicit, sp will point at least 256 bytes above it (see ***6.3 The Stack*** on page 6-4).
- fp contains 0 or points to the most recently created stack backtrace structure (see ***5.2.5 The stack backtrace data structure*** on page 5-6).
- 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 ***6.3 The Stack*** on page 6-4).
- Arguments are marshalled as described below.

6.4.2 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 a:

- word-sized integer value, or
- 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.)

6.4.3 Control return

When the return link value for a function call is placed in the pc:

- sp, fp, sl 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 ***6.5 C Language Calling Conventions*** on page 6-7).
- If the function returns a simple floating-point value, the value is encoded in a1, a2 and a3.

6.5 C Language Calling Conventions

6.5.1 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.

The C compiler widens arguments of type float to type double to support interworking between ANSI C and classic C.

Char, short, pointer and other integral values occupy one word in an argument list. Char and short values are widened by the C compiler during argument marshalling.

Characters are naturally unsigned: ANSI C has signed, unsigned and plain chars.

6.5.2 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 (so 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.

6.5.3 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 sub-fields 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, ...);
```

6.6 Function Entry

6.6.1 Introduction

A complete discussion of function entry is complex. This section discusses a few of the most important issues and special cases.

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 *In general, tail continuation is difficult with the Thumb instruction set because of the limited range of the B instruction (+/-2048 bytes).*

6.6.2 Simple function entry

The entry sequence for functions is:

```
PUSH {save-registers, lr} ; Save registers as needed.
```

The 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 function exit sequence becomes:

```
POP {save-registers}
POP {a3}
ADD sp, sp, #16
MOV pc, a3
```

6.6.3 Function entry: 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 at least on function entry. The stack must be checked when:

- the function uses 256 bytes or less of stack space
- the function uses more than 256 bytes of stack space, but the amount is known and bounded at compile time

The TPCS does not support languages in which the amount of stack required for a function is known only at runtime. This is not a requirement for languages that support open array arguments (such as Modula-2 and Pascal), since the arguments can be placed in the callee stack frame where the size is known.

The check for stack limit violation is made at the end of the function entry sequence. Register ip is available as a work register. If the check fails, one of the following standard runtime support functions is called:

- `__rt_stkovf_split_small`
- `__rt_stkovf_split_big`)

Each environment that supports explicit stack limit checking must provide these functions, which can:

- terminate execution, or
- extend the existing stack, decrementing sl

6.6.4 Stack limit checking: small, fixed frames

For frames of 256 bytes or less, the limit check may be implemented as follows:

```
    CMP    sp, sl
    BGE    no_ovf
    BL     |__16__rt_stkovf_split_small|
no_ovf
```

6.6.5 Stack limit checking: large, fixed frames

For frames larger than 256 bytes, the limit check may be implemented as follows:

```
    LDR    ip, framesize
    ADD    ip, sp
    CMP    ip, sl
    BGE    no_ovf
    BL     |__16__rt_stkovf_split_big|
no_ovf
    ...
    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.*

6.7 Function Exit

To exit from functions, use:

```
MOV pc, lr
```

where `lr` has the same value as it had on entry to the function. `lr` does not need to be preserved.

To exit from functions which 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       ; push-args only valid
                                    ; if variadic
MOV pc, a4                          ; Return
```

7 Toolkit Utilities

This chapter describes the software utilities provided with the Toolkit.

7.1 Introduction

This chapter describes the software utilities provided with the *Software Development Toolkit*:

ARM Profiler	Displays an execution profile of a program from a profile data file generated by either the windowed debugger or by armsd.
ARM Librarian	Allows sets of related 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.
ARM Object Format Decoder	Decodes AOF files such as those produced by armasm and armcc.
ARM Executable Format Decoder	Decodes executable files such as those produced by armasm and armcc.
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.

7.2 ARM Profiler

The ARM Profiler, armprof, displays an execution profile of a program from a profile data file generated by either the windowed debugger or by armsd. 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 which 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.

No special options are needed at compile time to allow profile data to be generated for a program, nor is it necessary to take any special action at link time (other than ensuring that the program image contains symbols, as is the linker default).

Profiling is available only for programs loaded into store by the debugger; function call counting is not available for code in ROM. If function call counts are required, the debugger must be informed when the program image is loaded (and it alters the program, diverting calls to counting veneers).

The debuggers allow the collection of PC samples to be turned on and off at arbitrary times, allowing data to be generated only for the part of a program on which attention is focused (omitting initialization code, for example).

Notes *Take care that the time between turning sampling on and off is long by comparison with the sample interval, or the data generated may be meaningless.*

Turning sampling on and off does not affect the gathering of call counts.

7.2.1 Profiler command-line options

A number of options are available to control the format and amount of detail present in the profiler output.

`-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.
`-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.
`-noparent`	Turns off the parent listing.
`-nochild`	Turns off the child listing.
`-sort cumulative`	Sorts the output by the total time spent in each function and all of its children.

`-sort self`	Sorts the output by the time spent in each function (excluding the time spent in its children).
`-sort descendants`	Sorts the output by the time spent in all of a function's children but excluding time spent in the function itself.
`-sort calls`	Sorts the output by the number of calls to each function in the listing.

By default, child functions are listed, but not parent functions, and the output is sorted by cumulative time.

Example

```
armprof -parent sort.prf
```

7.2.2 Profiler 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 just a single line.

The following shows example 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
----------------------------------------------------------------
```

Functions listed before the current function are parents of that function, and functions listed afterwards are child functions.

`cum%` Is only applicable to the current function and gives the percentage of the total time accounted for by this function and all of its children.

The other columns have slightly different meanings depending on whether the line is a parent function, a child function or the current function itself.

For the current function:

`self%` Gives the percentage time spent in this function itself.

`desc%` Gives the percentage time spent in the children of this function.

`calls` Gives the number of calls to this function.

For a parent function:

`self%` Gives the percentage time spent in the current function itself on behalf of this parent.

`desc%` Gives the percentage time spent in the children of the current function of behalf of this parent.

`calls` Gives the number of calls made by this parent to the current function.

For a child function:

`self%` Gives the percentage time spent in this child on behalf of the current function.

`desc%` Gives the percentage time spent in this child's children on behalf of the current function.

`calls` Gives the number of times this child was called by the current function.

7.3 ARM Librarian

The ARM Librarian (armlib) allows sets of related 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 will be eliminated if the output is AIF or if the `-nounusedareas` 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 ***3.4 Area Placement and Sorting Rules*** on page 3-13.

The full specification of ARM Object Library Format can be found in ***Chapter 13, ARM Object Library Format***.

7.3.1 Librarian command-line options

The format of the armlib command is:

```
armlib options library [file-list | member-list]
```

The wildcards * and ? may be used in `file-list` and `member-list`.

`options` can be any of the following:

`-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`*.
`-e`	Extracts members in *`member-list`*, placing them in files of the same name.
`-o`	Adds an external symbol table to an object library.
`-l`	Lists library. This may be specified together with any other option.
`-s`	Lists symbol table. This may be specified together with any other option.
`-v` *`file`*	Reads in additional arguments from a file, in the same way as the armlink *`-via`* option, described on page 3-5.

7.4 ARM Object Format Decoder

The ARM Object Format (AOF) file decoder, decaof, is a tool which decodes AOF files such as those produced by armasm and armcc. The full specification of AOF can be found in ***Chapter 14, ARM Object Format***.

7.4.1 Object file decoder command-line options

The format of the `decaof` command is:

```
decaof [-options] file [file ...]
```

`options` consists of a string of letters, which have the following meaning:

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 or help	Gives online details of the decaof command.
q	Gives a quick report of the area sizes only.
r	Prints relocation directives (and implicitly includes `-d`).
s	Prints symbol tables.
t	Prints string tables.
z	Prints a one-line code and data size summary per file.

If no options are specified, the effect is of `-dst`.

Each file should be an AOF file, otherwise `decaof` will complain.

Example

```
decaof -q test.o
C$$code  4748
C$$data  152
```

7.5 ARM Executable Format Decoder

The ARM Executable Format (AXF) file decoder, decaxf, is a tool which decodes executable files such as those produced by armlink.

7.5.1 Executable file decoder command-line options

The format of the `decaxf` command is:

```
decaxf [-options] file [file ...]
```

`options` consists of a string of letters, which have the following meaning:

- `c` disassembles code
- `g` prints debug areas
- `s` prints the symbol table

The following *`options`* are for ELF files only:

- `t` prints the string table
- `d` displays the contents of the data section
- `r` displays relocation information

Each file should be an executable file, otherwise `decaxf` will complain.

Examples

```
decaxf -gst my_elf.axf
decaxf -c test1.axf test2.axf test3.axf
```

7.6 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, primarily by rewriting top-level function prototypes (whether declarations or definitions).

The topcc translator performs its translation prior to the C preprocessing phase of any following compilation, and ignores preprocessor flag settings. It is therefore unable to help with the translation of sources where function prototypes have been obscured, for example, by preprocessor macros.

The translation performed 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.

7.6.1 ANSI to PCC C command-line options

The command format for topcc is:

```
topcc options [infile [outfile]]
```

where:

infile defaults to stdin

outfile defaults to stdout

The options are as follows:

`-d` describe what the program does

`-c` do not remove keyword `const`

`-e` do not remove `#error...`.

`-p` do not remove `#pragma...`.

`-s` do not remove keyword `signed`

`-t` do not remove 2nd argument to `va_start()`

`-v` do not remove keyword `volatile`

`-l` do not add `#line` directives

7.6.2 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

1 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>`.

2 ANSI keywords `const`, `signed`, and `volatile` are removed (with warnings), and enums are warned of (stricter usage under PCC).

3 Type `void *` is converted to `VoidStar`, which should be typedef'd to `'char *'` to be compatible with PCC.

4 ANSI C's unsigned and unsigned long constants are rewritten using the typecasts `(unsigned)` and `(unsigned long)`. (For example, `300ul` becomes `(unsigned long)300L`.)

5 After rewrites that change the number of lines in the file, `#line` directives are included that resynchronize line numbering. These quote the source filename, so that debugging tools then refer to the ANSI form of sources.

7.6.3 Issues with topcc

1 topcc takes no account of the setting of conditional compilation options. This is quite 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.
```

2 topcc cannot concatenate adjacent string literals. In practice, all important uses of ANSI-style implicit concatenation involve some mix of literals and preprocessor variables (of which topcc is oblivious). topcc could easily concatenate adjacent string literals.You can eliminate these from the input program beforehand.

3 If topcc finds an extra closing brace and starts processing text prematurely as if it were at the top level, this 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, most of topcc's other transliterations can be inhibited to make these principal changes more visible. (See ***7.6.1 ANSI to PCC C command-line options*** on page 7-9.*)*

Debug Reference

8 Angel

This chapter contains information about Angel, the ARM Debug Monitor. To aid readability, the term *Angel* is used to mean *the Angel Debug Monitor* throughout this chapter.

8.1 Introduction

Angel (the Angel Debug Monitor) is a program which allows rapid development and debugging of applications running on ARM-based hardware. Angel can debug applications running in both ARM and Thumb state on target hardware.

For an overview of Angel, and an introduction to the various components of an Angel system, refer to the *Software Development Toolkit User Guide (ARM DUI 0040)*.

8.2 Structure

The main components of Angel are shown in ***Figure 8-1: Structure of Angel***.

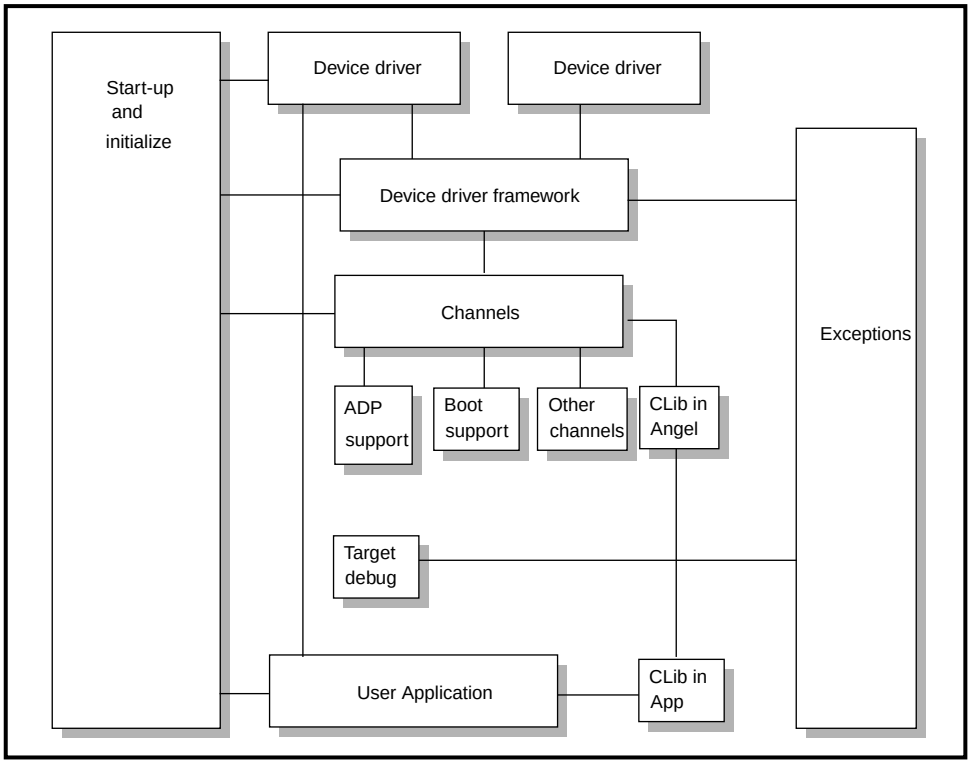


Figure 8-1: Structure of Angel

The C Library is split into:

- the C Library itself, which is linked with the application
- support for the semihosted parts of the C library which is linked with Angel

8.3 Angel C Library Support (SWIs)

Angel uses a SWI mechanism to allow user applications linked with the ARM C Library for Angel to make semihosting requests, eg. requests such as ‘open a file on the host’, which have to be communicated to the host in order for them to be carried out.

Angel’s predecessor, the Debug Monitor (Demon), also supported SWIs for this purpose. However, Demon used a fixed set of SWIs in the range 0 to 0x71, and sometimes this was found to be inconvenient because it made it difficult for operating systems developers to make use of SWIs in this range for their own purposes.

Therefore Angel uses only a single SWI, which can be configured (see `angel_SWI_*` in `arm.h`) to be any SWI, so that it need not clash with any SWIs that operating system developers wish to make use of. However, this means that the operation type must be passed in r0, rather than being encoded as part of the SWI number. All other parameters are passed in a block which is pointed to by r1. The result is returned in r0 either as an explicit return value or as a pointer to a data block. See the description of each operation below.

This interface is different and incompatible with the Demon SWI interface, although many of the SWIs still perform the same functions.

For users of the Angel C Library this does not matter—use of the new SWI mechanism will be transparent. However, assembler programmers who have written code which makes use of the Demon semihosting SWIs must recode the calls to the Demon SWIs to make use of the new Angel SWI interface.

Assembler programmers should also note that r1–r3 are preserved by Angel when an Angel system call is made. However, r0 is used to return a result: if no result is returned, it will have been corrupted.

8.3.1 The semihosted operations

In the following descriptions, the number in parentheses after the operation’s name (for example 0x01) is the value r0 should be set to for this operation. In general, it is best to use the operation names which are defined in `arm.h`.

SYS_OPEN (0x01)

r1 addresses a pointer to an argument block. The first word is a pointer to a NULL-terminated string containing a file or device name. The second word is a small integer which specifies the file opening mode. The third word is an integer which gives the length of the string (not including the NULL).

If the open succeeds, a non-zero handle is returned in r0, otherwize zero is returned in r0.

mode	0	1	2	3	4	5	6	7	8	9	10	11
ANSI C fopen mode	r	rb	r+	r+b	w	wb	w+	w+b	a	ab	a+	a+b

SYS_CLOSE (0x02)

On entry, r1 must be a pointer to a handle for an open file, previously returned by SYS_OPEN. If the close succeeds, zero is returned in r0; otherwise, a non-zero value is returned.

SYS_WRITEC (0x03)

Writes a byte, pointed to by r1, to the debug channel. When executed under the symbolic debugger, the character appears on the display device connected to the debugger.

SYS_WRITE0 (0x04)

Writes the null-terminated string to the debug channel. On entry, r1 contains a pointer to the string. When executed under the symbolic debugger, the characters appear on the display device connected to the debugger.

SYS_WRITE (0x05)

On entry, r1 contains a pointer to a data block, the first word of which must contain a handle for a previously opened file. The second word points to a buffer, and the third word contains the number of bytes to be written from the buffer to the file. SYS_WRITE returns, in r0, the number of bytes not written (and so indicates success with a zero return value).

SYS_READC (0x06)

Reads a byte from the debug channel, returning it in r0. The read is notionally from the keyboard attached to the debugger.

SYS_READ (0x07)

On entry, r1 contains a pointer to a data block, the first word of which must contain a handle for a previously opened file. The second word points to a buffer, and the third word contains the number of bytes to be read to the buffer from the file. SYS_READ returns, in r0, the number of bytes not read, and so indicates the success of a read from a file with a zero return value. If the handle is for an interactive device (SYS_ISTTY returns non-zero for this handle), a non-zero return from SYS_READ indicates that the line read did not fill the buffer.

SYS_ISERROR (0x08)

On entry, r1 must contain a pointer to a data segment, the first word of which is the required status word to check. If the status word in negative, SYS_ISERROR returns a non-zero value in r0. If the word is positive, SYS_ISERROR returns a zero value.

SYS_ISTTY (0x09)

On entry, r1 must contain a pointer to a handle for a previously opened file or device object. On exit, r0 contains -1 if the handle identifies an interactive device and 0 otherwise. Any other value indicates an error condition.

SYS_SEEK (0x0A)

On entry, r1 is a pointer to a data block, the first word of which is a handle for a seekable file object, the second word is the absolute byte position to be sought to. If the request can be honoured, SYS_SEEK returns 0 in r0; otherwise it returns a host-specific non-zero value. Note that the effect of seeking outside of the current extent of the file object is undefined.

SYS_FLEN (0x0C)

On entry, r1 is a pointer to a data block, the first word of which is a handle for a previously opened, seekable file object. SWI_FLEN returns, in r0, the current length of the file object, otherwise it returns -1 to indicate an error.

SYS_TMPNAM (0x0D)

On entry, r1 points to an argument block, the first word is a pointer to a buffer, the second word a target identifier for this filename. The third word contains the length of the buffer (length should be at least the value of L_tmpnam on the host system). On successful return, r0 returns 0 and the buffer contains the filename.

SYS_REMOVE (0x0E)

Deletes the file named by the nul-terminated string addressed by the first word of the argument block pointed to by r1; the second word is the string's length. Returns (in r0) a zero if the removal succeeds, or a non-zero, host-specific error code if it fails.

SYS_RENAME (0xF)

r1 contains a pointer to a data block, the first word of which is a pointer to the name of the old file. The second word is the length of the old filename. The third word is a pointer to the name of the new file name and the fourth is the length of the new name. Both strings are NUL terminated. If the rename succeeds, zero is returned in r0; otherwise, a non-zero, host-specific error code is returned.

SYS_CLOCK (0x10)

Returns, in r0, the number of centiseconds since the support code started executing. Note that by default, Angel will ask the host for this information, which can make it unreliable or inaccurate for some benchmarking purposes.

SYS_TIME (0x11)

Returns, in r0, the number of seconds since the start of 1970.

SYS_SYSTEM (0x12)

Passes the string, pointed to by the first word of the argument block pointed to by r1, to the host's command line interpreter, the string's length is contained in the second word of the argument block. The return status is returned in r0.

SYS_ERRNO (0x13)

Returns, in r0, the value of the C library `errno` variable associated with the host support for this debug monitor. `errno` may be set by a number of C library support SWIs, including SYS_REMOVE, SYS_REMOVE, SYS_OPEN, SYS_CLOSE, SYS_READ, SYS_WRITE, SYS_SEEK, and so on. Whether or not, and to what value `errno` is set is completely host-specific, except where the ANSI C standard defines the behavior.

SYS_GET_CMDLINE (0x15)

Returns a string of the command line used to call the executable; this is returned in the address pointed to by r1. On entry, r1 must contain a valid pointer. The returned NULL terminated string can be any length up to the default buffer size (minus the transport overhead); the buffer must be able to contain this. An error code may be returned in r0.

SYS_HEAPINFO (0x16)

The only parameter in the block pointed to by r1 is a pointer to a block of four words of data in which this SWI will return the stack and heap base and limit as follows:

1st word: Heap Base

2nd word: Heap Limit

3rd word: Stack Base

4th word: Stack Limit

The values returned are typically used by the Angel C Library during initialization.
The values returned by this are those values set up in `devconf.h`. However, the C Library can override these values, but will only do so if `__heap_base` is defined at link time, in which case the values of the following symbols will be used: `__heap_base`, `__heap_limit`, `__stack_base`, `__stack_limit`.

Note that this call will return sensible answers if EmbeddedICE is being used—but the values will be determined by the host debugger. For example, using EmbeddedICE versions 2.00 onwards, the variable `$top_of_memory` can be used to tell EmbeddedICE what to return as the top of memory. If this is not flexible enough, the exact addresses of these four items can be set by defining symbols as described above.

8.3.2 Other operations

There are three other operations which are intended for use by the user's application:

`angel_SWIreason_EnterSVC`

The Angel SWI should be called with r0 set to `angel_SWIreason_EnterSVC`.

This will return in SVC mode with interrupts disabled. On return, r0 holds the address of a routine which should be called to return to USR mode, `Angel_ExitToUSR`. This is so that standalone programs can make use of this facility.

`angel_SWIreason_LateStartup`

The Angel SWI should be called with r0 set to `angel_SWIreason_LateStartup`, and with r1 set to AL_CONTINUE or AL_BLOCK as defined in `support.h`.

This is used to tell Angel that an application linked with a late-startup Angel library now requires debugger support.

`angel_SWIreason_ReportException`

The Angel SWI should be called with r0 set to `Angel_SWIreason_ReportException`, and with r1 set to one of the values defined in `adp.h` with names starting `ADP_Stopped_`.

This SWI allows the application to report an exception to the debugger directly. The most common use is for reporting that execution has completed, and the value to use is `ADP_Stopped_ApplicationExit`.

8.4 ROM Applications and Late Debugger Start-up

This section describes how the debugger and the target interact, and describes how Angel asks for a program to be started without handing control to the debugger.

8.4.1 Flow of control

The ARM debuggers assume they are in control of the target system. When they start up, they work on the basis that the user application is not executing and has been halted.

The ARM debuggers then look around the target system to find out what is in the registers and some regions of memory, before handing control directly to the user of the debugger. The target stays 'halted' until the user of the debugger tells the debugger to restart execution. When this is done the debugger relinquishes control.

The target then resumes (or starts) the target application until a breakpoint or semihosting request is encountered. It is also possible for the user of the debugger to request that the target stops by issuing an interrupt request, eg. by typing Ctrl C from armsd. When any of these (except the semihosting request) happen, the debugger takes back control.

Control for applications in ROM

If the application has to be downloaded (as it is not in ROM), it is not important that the debugger takes control on startup. If the application is already in ROM (in later stages of development), it may not be desirable for the debugger to take control immediately.

8.4.2 Late debugger startup

A SWI operation allows the application to indicate that it is ready for a debugger to take control. When this is called, the application stops executing and sets a flag to indicate that a debugger may now attach itself.

A debugger can then attach to Angel and interrogate the state of the application. If a debugger attempts to attach to the target *before* the target has called this SWI, Angel does not allow the debug session to start, and refuses to service any requests made by the debugger. This allows late debugger startup systems to be implemented, where the application executes immediately after Angel has booted, and a debugger cannot interfere. However, if an error condition is detected either by Angel or by the application, the debugger start-up SWI can be called and the debugger can then take control and inspect the system. You cannot make semi-hosting requests to the host before calling the debugger startup SWI, because there is no debugger attached to service the request.

An alternative form of late debugger startup is supported, where the late debugger startup SWI is called during application startup. The debugger then takes control, but the application can be restarted by typing `go`. This is useful for systems which require semihosting support and also have the application in ROM.

To make your application work like this, define this symbol in your application:

```
__do_late_debug_swi_on_startup
```

The Angel C library notices this and takes appropriate action.

8.5 Breakpoints and Undefined Instructions

8.5.1 ARM state

In ARM state, there is one SWI, which can be reconfigured to any of the SWI numbers, so that it need not clash with the system developer's use of SWIs. This SWI is used for:

- system privilege
- semihosting requests
- reporting an exception to the debugger

In all cases, one or more registers are available to pass information to Angel.

There are also two undefined instructions:

`0xE7FDDEFE` for little-endian systems

`0xE7FFDEFE` for big-endian systems

When executed in Thumb state, the undefined instruction vector is still taken, whether executing the instruction in the top or bottom half of the word (in both cases these disassemble to a Thumb-undefined instruction and a branch to the Thumb-undefined instruction).

These are used for normal, user interrupt, and vector hit breakpoints. In all cases, there are no arguments in registers, but the breakpoint address itself is significant.

8.5.2 Thumb state

In Thumb state, there is one SWI, which can be reconfigured to any of the limited (256) SWIs available in Thumb state, so that it does not clash with the SWIs being used in your system. This SWI is used for semihosting requests; it may also be needed for gaining system privilege and reporting exceptions to the debugger, but only if the parts of Angel which do this are compiled into Thumb code, which is unlikely to be necessary.

There is also one undefined instruction; `0xDEFE`. This is the same undefined instruction as that used by EmbeddedICE.

Breakpoints

Note: *Although ARM breakpoints are detected in Thumb state, Thumb breakpoints are not detected in ARM state. It is not possible for a Thumb breakpoint to be detected if it is executed in ARM state, because the rest of the ARM instruction is not easily determined.*

The Thumb SWI is used for normal, user interrupt, and vector hit breakpoints. In all cases, there are no arguments in registers, but the breakpoint address itself is significant.

8.6 Communications Architecture for Angel

8.6.1 Layers

Figure 8-2: Communication layers for Angel is a conceptual model of the communication layers for Angel. (*ADP = HADP (host to target) or TADP (target to host), and so on). In practice, some layers may be combined.

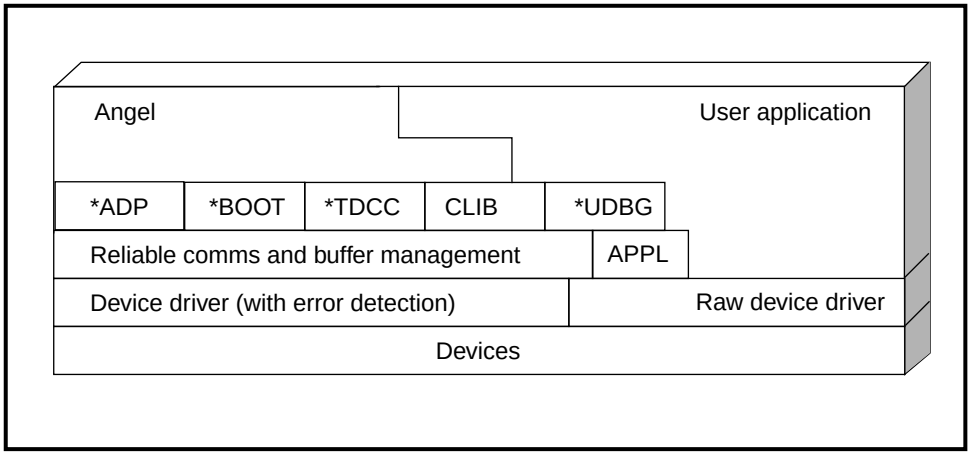


Figure 8-2: Communication layers for Angel

At the top level on the target, the Angel agent communicates with the debugger host (*ADP, *BOOT), and the user application may wish to make use of semihosting support (CLIB) or extended debugger features (*UDBG). The user application may use the device connected to the debugger host for its own communications (APPL) and/or may use other devices.

All communications for debugging (*ADP, *BOOT, *TDCC, CLIB, *UDBG) require a reliable channel between the target and the host. The Reliable comms and buffer management layer is responsible for providing reliability, retransmissions and multiplexing/de-multiplexing for these channels. Because achieving reliability after errors requires retransmission, this layer must also handle buffer management.

Raw packet access to the device driver by the application is multiplexed with reliable packets by the device driver. This minimizes the impact of removing Angel code, which can be replaced with a small ‘shim’ layer to a raw device driver.

The device driver layer provides detection or rejection of bad packets but does not offer reliability itself.

8.6.2 Channels

The channels used by Angel are described in ***Table 8-1: Angel communication channels***.

Channel	Type of data transmitted
HADP	ADP, host originated
TADP	ADP, target originated
HBOOT	Boot agent channel, host originated
TBOOT	Boot agent channel, target originated
CLIB	Semi hosting C library support, target originated
HUDBG	User Debug Support, host originated
TUDBG	User Debug Support, target originated
HTDCC	Thumb direct comms channel, host originated
TTDCC	Thumb direct comms channel, target originated
TLOG	Target logging to assist development, target originated

Table 8-1: Angel communication channels

8.6.3 BOOT support

If there are two (or more) possible debug devices (eg. serial and serial/parallel), the boot agent must be able to receive messages on any device and then ensure that further messages which come through the channels layer are sent to the correct (new) device.

When the debug agent detects a Reboot or Reset message, it opens the other channels using the device which received the message. All debug channels switch to use the newly selected debug device.

During debugging, each channel is connected via the same device to one host. Initially, Angel listens on all Angel-aware devices for an incoming boot packet, and when one is received, the corresponding device is selected for further Angel use. To cope with host-end problems or restarts, the ability to listen on all devices for a boot message is maintained throughout a debugging session.

To support this, the channels layer provides a function to register a read callback across all Angel-aware devices, and a function to set the default device for all other channel operations.

8.7 Reliability and Retransmission

This section describes the buffer management and reliable communications layer.

8.7.1 Packets

There are two types of packet:

- the data packet
- the renegotiation packet

In an ideal situation, only data packets are used because each data packet is replied to with another data packet.

Renegotiation packets request resending of a previous data packet because of a lost or corrupted packet. If a packet is corrupted, a resend message is sent out and the bad packet is resent. If the resend itself is corrupted, another resend is sent and so on, until the link is re-established. (Resent, duplicated data is harmless and can be thrown away but lost data should be avoided as it is difficult to recover from.)

8.7.2 Transmission sequencing

If the two ends of the link get out of step, the sequence numbers are used to calculate which packets need to be resent in order to resynchronize the ends. A missing message blocks the protocol until another message is sent out. This causes another resend request because the sequence numbers are wrong.

The protocol used in Angel uses two sequence numbers in each packet to indicate how many messages each has transmitted:

- one sequence number for the host
- one sequence number for the target

The two sequence numbers are referred to as the *home* number for the node's own number and the *opposing* number for the number of the opposing node.

A two-letter convention describes the sequence numbers:

- the first letter is either:
  - `N` to indicate a generic node
  - `M` to indicate that the identifier originated in the message. This indicates the sequence numbers contained in the message.
- the second letter is either:
  - h for the node's home number
  - o for the opposing node's home number

  The h and o denominators are those of the originating node.

Retransmissions use the same number that they used the first time they were sent; all subsequent messages will continue from this number. All messages must be stored until they have been acknowledged, in case they need to be retransmitted.

8.7.3 Protocol

The rules for the protocol are:

Sending a new message
Increment `Nh` but leave `No` unmodified.
Send the message using these numbers.

Receiving a good message
Check that:

`Mh = No + 1`	Sequence numbers ok; increment `No`
`Mh > No + 1`	A message has been lost; ask for a resend using existing `Nh` and `No`
`Mh < No + 1`	Duplicate message already received; discard, leaving sequence numbers unchanged

Receiving a bad message
Send a resend message with the node's current sequence numbers.

Receiving a resend message
`Mo` and `Mh` indicate what the opposing node last received and sent, so check the values to see what went wrong:

`If Nh != Mo`	The opposing node has missed message `Mo+1`, so resend messages `Mo+1` onwards using the messages' original sequence numbers. The node numbers remain unmodified.
`Else If No != Mh`	The node missed message `Mo+1` so send a resend message using Nh and No.

8.7.4 State diagram for the protocol

Figure 8-3: State diagram shows the state diagram for the protocol.

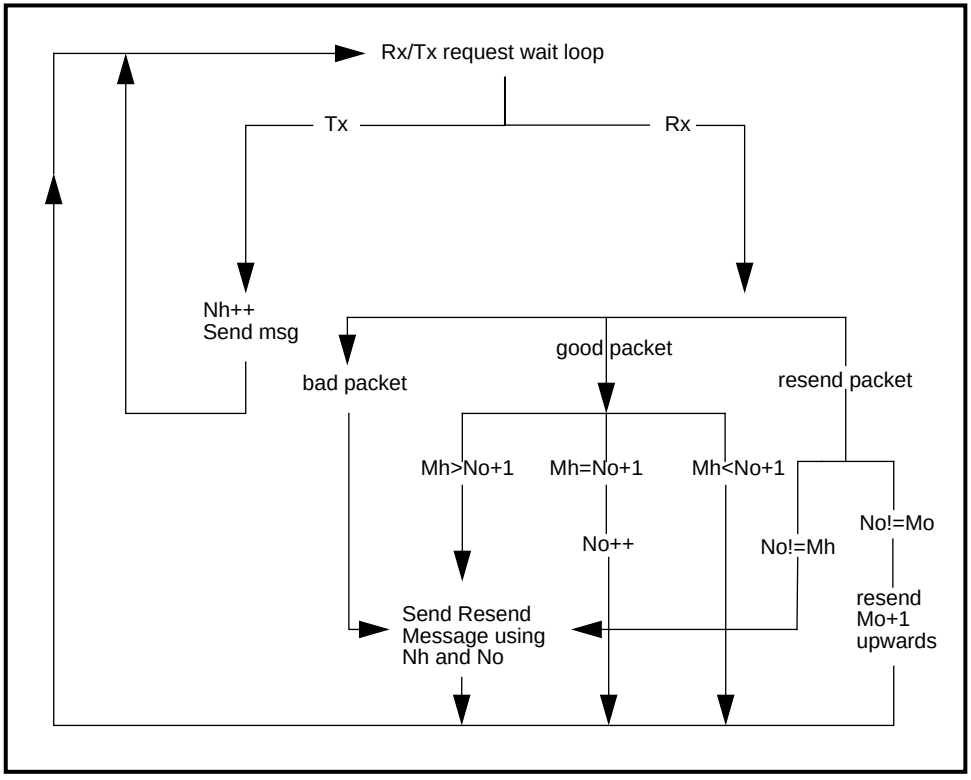


Figure 8-3: State diagram

As soon as Angel receives acknowledgement of a message, it can remove the message from the buffer. Until this time, the message must be stored in case of retransmission. A message is always acknowledged by another message.

A resend request is a special-purpose message and is identified as such, because it cannot usefully carry any data. On receiving a ‘good message’, Angel does not send out an acknowledgement immediately (because it can be acknowledged by the next data packet, unless the message was a resend request. In this case, Angel sends out the requested packet, or a renegotiation packet (a resend request) if the data is out of sequence.

8.7.5 Recovering from a lost packet

A lost packet causes a semideadlock situation which is resolved as soon as one of the ends has to send out another packet. This packet forces a resend of the original message. If another message is never sent, the deadlock situation occurs.

In order to ‘kick start’ a deadlock, a message is sent every few seconds in a heartbeat packet. This kick starts a deadlock which occurs if a missing packet is not followed by any further packets on other channels. See ***8.8.7 Heartbeat mechanism*** on page 8-18.

8.8 Channels Layer and Buffer Management

The channels layer is responsible for multiplexing the various Angel channels onto a single device, and for providing reliable communications over those channels. The channels layer is also responsible for managing the pool of buffers used for all transmission and reception over channels.

Because there are several channels that could be in use independently (eg. CLIB and HADP), the channel layer accepts only one transmission attempt at a time.

8.8.1 Channel restrictions

To simplify the design of the channels layer and to help ensure that the protocols operating over each channel are free of deadlocks, the following restriction is placed on the use of each channel.

For a particular channel, all messages must originate from either the Host or the Target, and responses may be sent only in the opposite direction on that channel. Therefore two channels are required to support ADP:

- one for host originated requests (Read Memory, Execute, Interrupt Request)
- one for target originated requests (Thread has stopped)

8.8.2 Buffer management

Managing retransmission means that the channels layer must keep hold of messages that have been sent, until they are acknowledged. The channel layer supplies buffers to channel users who want to transmit, and then marks transmitted buffers as busy until acknowledged.

One buffer could be required per channel for transmission in the single-threaded world. In addition, to permit reception on the boot channel across all Angel-aware devices, one buffer is required per device. The theoretical total buffer requirement is shown below, but in practice, significantly fewer buffers than this are available because of memory constraints:

```
(number of channels C) + (number of Angel-aware devices D)
```

The buffers contain a header area sufficient to contain channel number and sequence IDs, for use by the channels layer itself. Any spare bits in the channel number byte are reserved as flags for future use.

To control buffer allocation, each buffer is marked according to its state. For example:

- free
- allocated to user
- awaiting acknowledgement
- allocated to device

8.8.3 Long buffers

Most messages and responses are short (typically around 40 bytes), although some may be up to 200 bytes long. However, there are some situations where larger buffers would be useful. For example, if the host is downloading programs or configuration data to the target, a larger buffer size reduces the overhead created by channel and device headers, by acknowledgement packets and by the line turnaround time required to send each acknowledgement (for serial links).

Limited RAM

When RAM is unlimited, the easiest solution is to make all buffers large. Because RAM in an Angel system is not normally an unlimited resource, there is a mechanism which allows a single large buffer to be shared.

When the device driver has read enough of a packet to determine the size of the packet being received, it performs a callback asking for a suitably sized buffer. If a small buffer is adequate, a small buffer is provided. However, if a large buffer is needed, and the special large buffer is free, this one is allocated. If a large buffer is required, but is not available, the packet is treated as a bad packet, and a resend request results.

8.8.4 Channel packet format

Figure 8-4: Channel packet format shows the channel packet format.

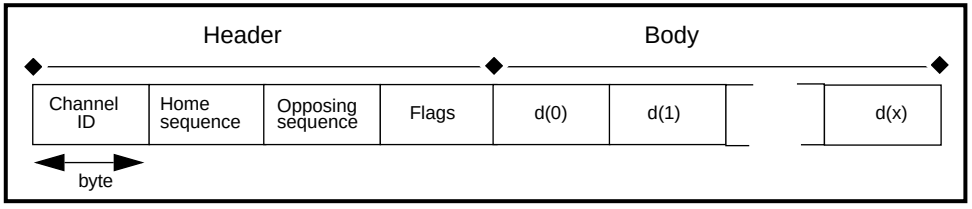


Figure 8-4: Channel packet format

The home sequence number is Mh, the sequence number of this message for a normal message. The opposing sequence number represents Mo, the highest in-sequence number received from the opposing side by the sender of the message. In other words, it is the acknowledgement sequence number. Note that this is a symmetrical approach independent of whether the sender is host or target, unlike the sequence numbers described in the existing `channels.h` header.

Initially three flags are defined:

`CF_RELIABLE = 1<<0`	reliable protocol in use
`CF_RESEND = 1<<1`	renegotiation (resend request) packet
`CF_HEARTBEAT = 1<<2`	heartbeat/keepalive packet (see ***8.8.7 Heartbeat mechanism*** on page 8-18)

Remaining bits are reserved for future use.

The length of the complete data packet is returned by the device driver layer. Since the channel header is fixed length, an overall length field for the user data portion of the packet it not required.

8.8.5 Channel buffer format

A word is reserved at the head of the buffer for internal use, specifically to permit the construction of linked lists of unacknowledged buffers per device. This reduces the useful buffer space by one word. If this is unacceptable, the default buffer size can be increased. It is more efficient to steal space from the buffer than to maintain a separate list within the channels layer.

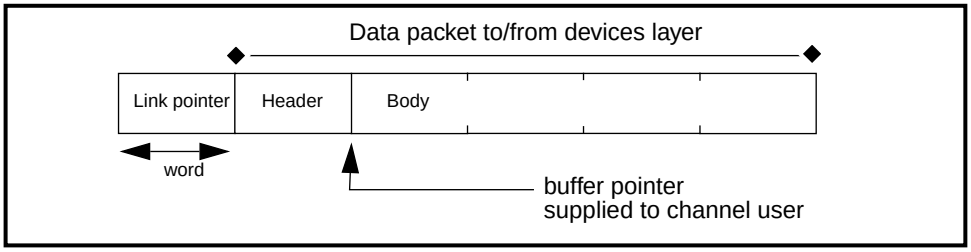


Figure 8-5: Channel buffer format

8.8.6 Buffer life cycle

The user of a channel must explicitly allocate a buffer before requesting a write. Buffers must be explicitly released *either* by passing the buffer to one of the channel transmit functions *or* by explicitly releasing it with the release function.

The channels layer supplies buffers containing received packets, and these too must be explicitly released *either* by the release function *or* by filling the buffer and passing it back to channels for transmission.

8.8.7 Heartbeat mechanism

Note *The implementation of heartbeats has changed at revision 2.11. The ADP timer now writes packets using at least the heartbeat rate, and uses heartbeat packets to ensure this. It expects to see packets back using at least the packet timeout rate, and signals a timeout error if this is violated. The new mechanism is only used when the revision 2.11 debugger, Angel RDI and Monitor software are used.*

In some circumstances—such as a lost packet in an Ethernet-based system—the target and host can become live-locked. This problem can be solved by another packet arriving and forcing a renegotiation sequence to take place.

A special heartbeat packet is sent, with a flag set to indicate it is a heartbeat packet. This forces a renegotiation sequence to take place and reestablish the link. These packets are sent out only by the host, and are never replied to: a resend request or a resend forms the

reply, if needed. The target ignores the packet if the sequence numbers are correct. The heartbeat packets themselves do not have their sequence numbers incremented and as such are a special case.

If the frequency of heartbeat messages is altered, it affects the speed of the link and, if set to too short a period, prevents any real packets from being sent. If the heartbeat rate is too slow, on a bad link the effect of lost packets is far greater. The heartbeat mechanism comes into effect only once the boot sequence has completed, so, should any packets go astray during this time, the boot fails and the debugger returns an error.

8.9 Device Driver Layer

Angel copes with polled and asynchronous devices, and with devices which start out in an asynchronous mode and end up polling the rest of a packet. At the top boundary of the device driver layer, the API need offer only asynchronous (by callback) read and write interfaces to Angel or the user's application. The Angel framework also makes use of the serializer, described in ***8.12 Serialization, Stacks and Modes*** on page 8-38.

8.9.1 Transmit and receive

Figure 8-6: Angel Framework structure shows the framework structure for receiving data:

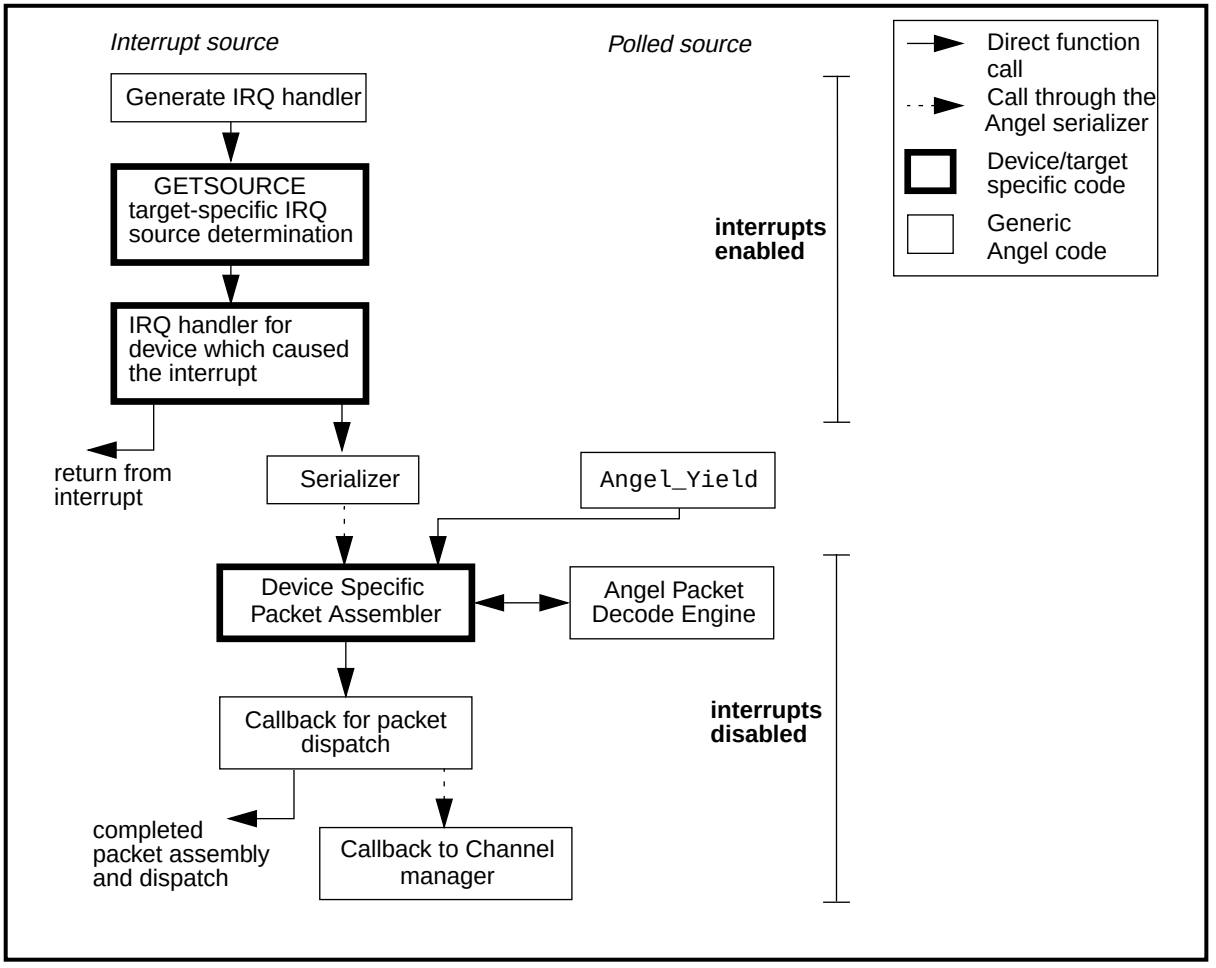


Figure 8-6: Angel Framework structure

Consider first a device driver which is interrupt driven (or which is currently using interrupts rather than polling). The sequence of events is described below:

1. When an interrupt occurs, the generic interrupt driver saves the state of the interrupted task and uses target-specific code to determine the source of the interrupt.
2. This source id is used to look up and call the function to process an IRQ for this device.
3. The interrupt handler for this device reads out any characters from the interrupting device and can then do one of two things:
   - return immediately
   - if enough characters have arrived to make it worthwhile, or if a part-interrupt, part-polled scheme is used, it can call the serializer with a request to execute code to process the data and/or poll in more.

   Note that interrupts are disabled up to this point. Therefore, the time taken to get to this step must be kept to a minimum to keep interrupt latency low.
4. The serializer sets running the device-specific packet-assembly code with interrupts enabled. This code can use the Angel packet-decoding engine to perform packet-length checks and CRC checks.

   Once a packet is fully assembled, a call is made to the serializer to dispatch to the channel manager.
5. The channel manager then dispatches the packet to the packet handler for the appropriate channel.

Polled devices

To handle polled devices, the device-specific packet assembler is called by the `AngelYield()` function call. This must be placed in any tight loop, whether within framework or device-specific code, in Angel, or in an application that coexists with Angel and polled devices. If this is not done, it is possible for deadlocks to occur on polled devices. On each call, the poll handler calls any enabled device driver that is in polling mode.

If a hardware timer is available, the polling operation can be performed on a clock tick. In this way, the application need never explicitly yield control. Your application can share a hardware timer with Angel; there is a modifiable set of functions which implement this, so you can (statically) modify the use of a hardware timer.

For transmission, the situation will be very similar to that shown above for reception.

Note that raw (user-only) device drivers that are purely interrupt driven must fit in with the interrupt scheme described above. If there are user-only polled devices, your application must ensure that Angel or shared devices are also polled at appropriate intervals. To do this, call the `AngelYield()` function from within the application's own polling loop(s).

8.9.2 Angel packet decode engine

Character-oriented devices typically pass eight-bit characters one by one via some kind of connection, for example via a serial line or parallel line. Angel needs to send packets consisting of eight-bit characters. The start and end of a packet are marked with special characters (STX and ETX). Whenever a character within a packet happens to be a special character, it must be rewritten as a special escape character (ESC) followed by an encoded version of the original character (char | 0x40). Furthermore, the device may interpret other characters in a special way or use them for flow control. These characters must be escape-encoded too.

On top of the STX and ETX packet markings, the type of data, length of data and a CRC error check are added to the data. These fields are determined by their position in the packet:

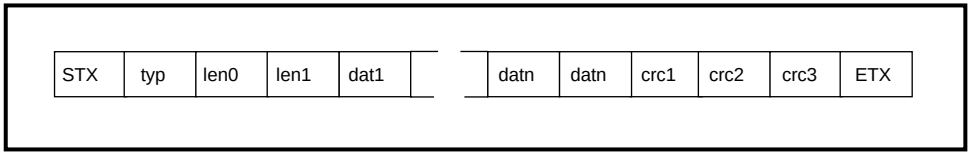


Figure 8-7: Angel packets

where n is the 16-bit length represented by len0 and len1.

The interpretation of packet layout, framing (STX,ETX), error detection (CRC) and transparency (ESC) is standard across all character-based devices, so support for this is factored out into a reception engine and a transmission engine.

The reception engine

Incoming characters are pre-processed to handle special characters and transparency. The results of this (events and clear 8-bit characters) are passed to a state machine which keeps track of the components that make up a packet, and assembles the relevant components into a buffer.

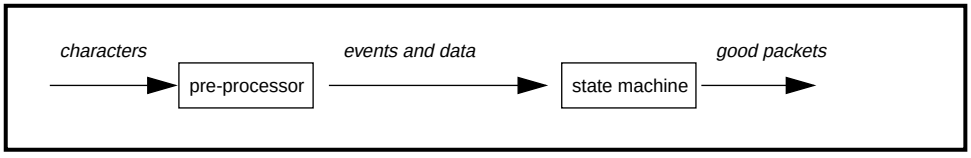


Figure 8-8: Reception engine

The device driver passes each character to the engine, along with a buffer and some state information which is manipulated by the engine. The engine assembles data into the buffer. Each time it is called, the engine returns a code indicating its current status:

- awaiting a packet
- processing a packet
- detected a bad packet
- received a good packet

When the engine indicates a good packet, the device driver can pass the completed buffer up to its client.

Transmission engine

There are four components to the transmission engine:

framer	responsible for writing type, length, data and CRC fields in correct order
escaper	escape-encodes any characters from the framer that need it
encapsulator	places STX and ETX around the output of the escaper
flow control	slips XON and XOFF characters into the output stream

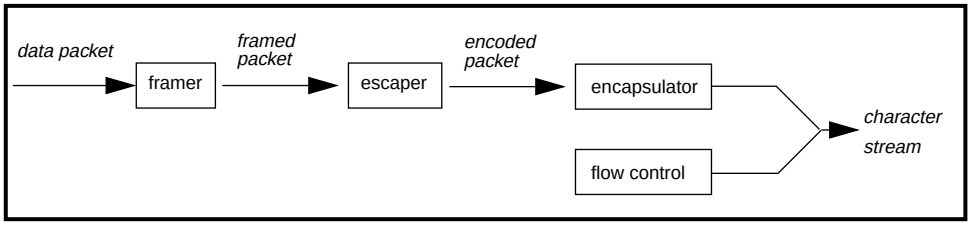


Figure 8-9: Transmission engine

Unfortunately, this has to be done character by character. One function is called by the device driver to set up the transmission engine with a new packet for transmission. A second function is called every time the device wants a new character to transmit.

The insertion of flow control characters into the outgoing character stream is carried out within the device-specific code, otherwise the transmission engine cannot easily initiate the transmission of a flow control character if it is in its idle state (ie. not in the middle of a packet).

8.9.3 Support for callback across all devices

This is primarily a channels layer issue, but to support the need of the BOOT channel to listen on all Angel-compatible devices, the channels layer needs to find out how many devices it should listen to for boot messages, and it needs to know which devices those are.

To provide this statically, the devices layer exports the appropriate device table or tables, together with the size of the tables.

8.9.4 Transmit queueing

As the core operating mode is asynchronous and more than one thread may be using a device, Angel rejects all but the first request, returning a ‘busy’ error message, and leaves the user (channels or the user application) to retry later.

8.9.5 Angel Interrupt handlers

Angel Interrupt handlers are installed statically (ie. at link time). The Angel Interrupt handler has been written to run off either IRQ or FIQ, although it is recommended that it should be run off IRQ. However, the ARM60 PIE board is configured by default to run using FIQs—and requires resoldering to make it run off IRQs—so the ARM60 ROM image runs off FIQ by default.

In order to change whether IRQ or FIQ is used, change the settings of `HANDLE_INTERRUPTS_ON_IRQ` and `HANDLE_INTERRUPTS_ON_FIQ`: it can work with both of these enabled.

The main difficulty with FIQs is that, on entry to the SWI handler and the Undefined instruction handler, FIQs are enabled, but should a FIQ happen at that point and end up calling `SerialiseTask`, the system fails. Therefore the interrupt handler, when built for use with FIQs, has special code to see if it is interrupting either of these handlers, and if this is so it postpones the FIQ.

Another piece of code which will need to be changed when adding a new device to the interrupt handlers is `GETSOURCE` in `target.s`. This needs to recognize the interrupt from the device. On a board such as the PID7T, where there is an interrupt controller, this needs to be queried (possibly for both FIQs and IRQs) before accessing the device's interrupt registers themselves.

For additional information about porting Angel, refer the *Software Development Toolkit User Guide (ARM DUI 0040)*.

8.9.6 Booting and parameter negotiation

The host and target attempt to establish contact following each power-up or hard reset. Initially the device drivers use default settings to communicate. If this first stage is successful the host and target negotiate the value of parameters (eg. baud rate) using the HBOOT and TBOOT channels, in the following stages:

1. The host sends a list of parameter settings that it can support to the target, using the default parameter setting for communication.
2. The target examines the list, selects appropriate settings, and returns these values to the host. If it cannot use the settings suggested by the host, it returns an error message.
3. The host switches to these new settings and sends a test message to the target, to which the target replies.

Parameter negotiation supports an arbitrary number of parameters. Initially there will be just one: serial baud rate. Negotiation is carried out by the host debugger and the target debug monitor via the HBOOT channel, so is not the responsibility of the device driver. However the device driver knows which parameters and options it supports, and is able switch to a parameter setting and reset to the defaults.

It is assumed that, for any device, there is a default configuration which will be selected by all hosts and targets on power-up or on a hard reset. This is essential for establishing initial contact.

Negotiation is always controlled by the host, which sends a `ParamNegotiate` request. This consists of a set of blocks, one block per parameter being negotiated. Within each block, the host lists the values it can operate at, starting with its preferred value. For serial baud rate, it might list 38400, 19200, 9600.

The target examines the list and chooses the best combination of parameters it can use, and sends a reply containing the same number of blocks as the request, but with a single value per parameter. Immediately after sending the reply, the target switches to the configuration it has chosen.

On receipt of a valid reply, the host switches to the configuration chosen by the target. To ensure that all is well, a `LinkCheck` message is then sent to the target, and the target replies. The link is now successfully established at the new settings.

If the target cannot select a configuration from the set offered by the host, it replies to the `ParamNegotiate` message with an error status, and the link remains at the default settings. The host can either continue at default settings or try another negotiation.

Figure 8-10: Parameter negotiation shows the details of the negotiation:

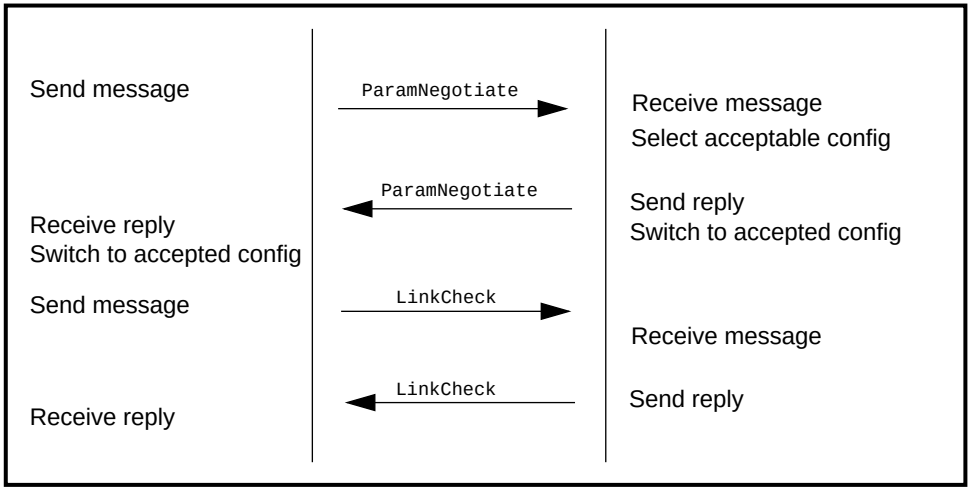


Figure 8-10: Parameter negotiation

8.9.7 Control calls

Angel-only or shared device drivers in an Angel-cooperating system provide a control entry point which supports the enable/disable transmit/receive commands, so that Angel can control application devices at critical times.

Angel device drivers provide control calls for:

- disabling and enabling the reception of data
- initializing the device
- resetting the device to its default state
- setting the device config to a set of specified parameters

These are implemented in a manner similar to the UNIX `ioctl` call. That is, there is a single entry point for these operations. The operation is determined by the value of the first parameter. This allows extra device-specific functions to be added without changing the device driver description.

8.9.8 The APPL device channel

Device drivers may perform multiplexing so that the device can be shared between the reliable channels layer and the user application. When Angel support is finally removed from the user system, changes to the application are minimal. The API for the APPL channel makes it easy for the developer to replace the Angel-specific drivers with a raw driver that services only the APPL API.

Although it can support a number of different device drivers, the host-side channels interface only has one device driver *active* at any given point in time. This interface's main purpose is to multiplex packets from the various ADP channels through the currently active device driver, both for transmit and receive.

A channel packet consists of four octets of channel header, followed by an indeterminate amount of opaque data. When a packet is passed down to the channels interface, the upper layers leave the first four octets free for the channel header to be written into them; similarly, received packets passed up from the channels interface preserve the channel header in the first four octets.

Angel supports multiplexed communications channels for user programs over the debug communications link. In other words, user applications also have direct access to the device drivers, rather than via the channels interface. This is achieved via a “device switching” veneer between the channels layer and the device drivers:

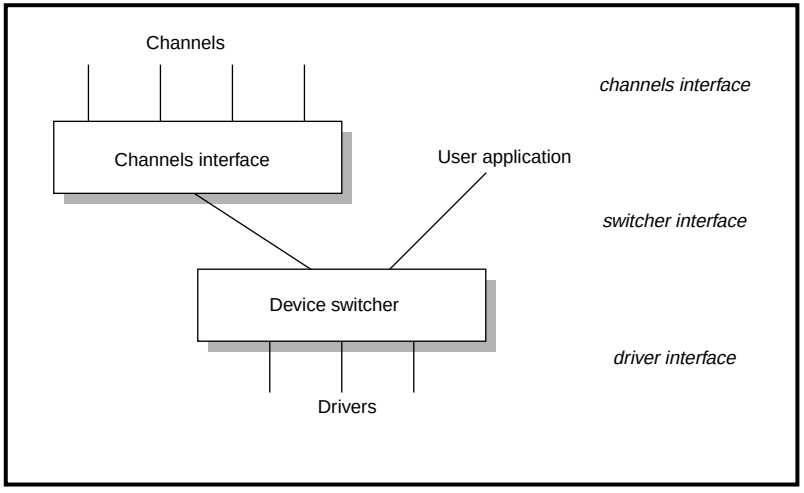


Figure 8-11: Host channel driver interface

8.10 Support for user application devices

The Angel framework provides user applications with a simple, unified interface to devices. This interface is specified in `devappl.h`, which provides blocking reads and writes, an `ioctl()`-like control call, and a yield call.

`devappl.h` hides the details of device driver access for three different cases:

- A — Full Angel, device shared with Angel for debugging communications *and* user application communications.
- B — Full Angel, device exclusively for use by applications (raw device). This case is possible only in systems with more than one device, because Angel requires shared use of at least one device.
- C — Minimal Angel, raw device.

It is relatively easy to migrate from case B to case C, because the raw device driver already exists and the Angel device can be removed once debugging is complete.

Moving from case A to case C requires changing from an Angel/shared device driver to a raw device driver. To ease this process, a framework is provided for character-oriented devices such as serial ports. The framework provides a packet-based skeleton for Angel/shared devices and a character-based skeleton for raw devices, both of which interface to the same low-level hardware-specific driver. Only this lower layer needs to be ported to new hardware, and it is used unchanged in all three cases.

8.10.1 Full Angel with one shared and one raw serial port

Figure 8-12: Full Angel with one shared and one raw serial port illustrates case B, where there are two ports (devices) provided by the same low-level device driver code, one of which is used as an Angel/shared device and the other as a raw device—for example, a PID card with two serial ports.

For case A, ignore the right-hand flow through `devraw.c` and `serraw.c`.

Case B is desrcibed in ***Figure 8-13: Minimal Angel with one raw serial port*** on page 8-31.

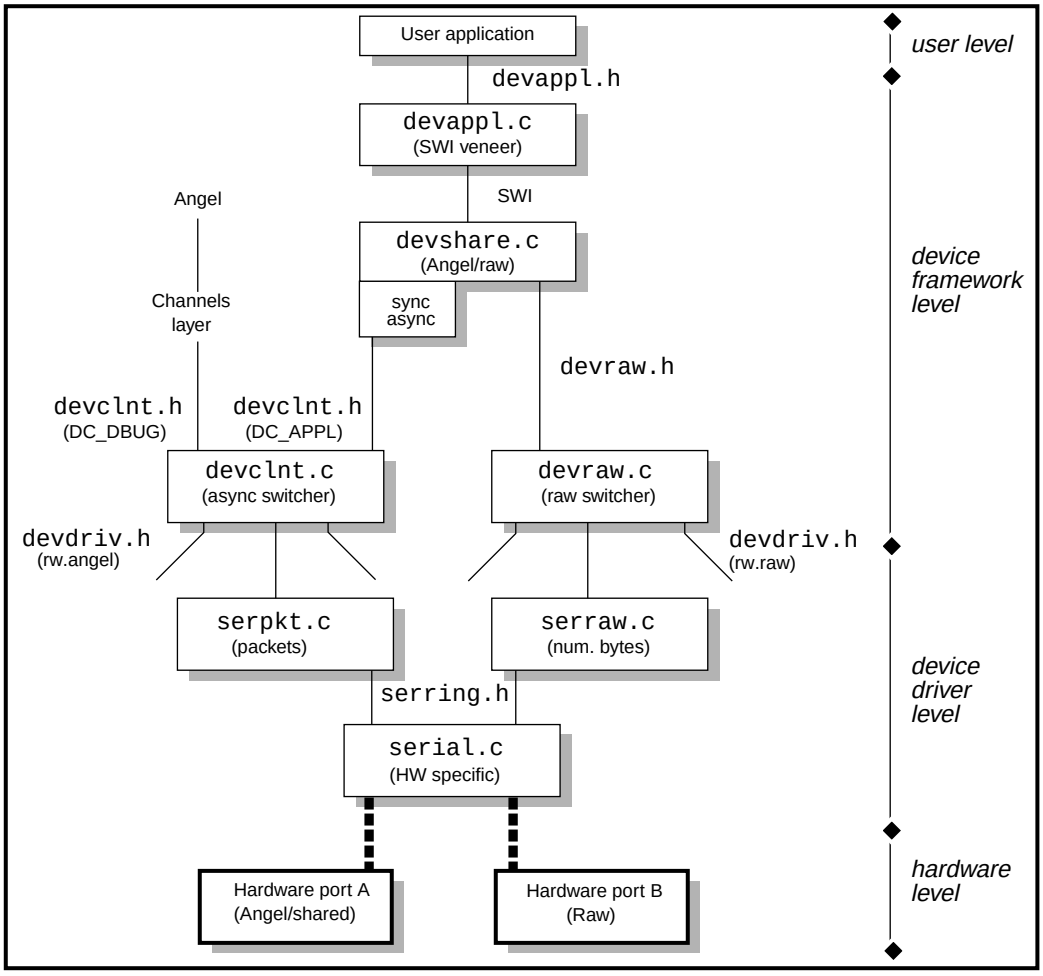


Figure 8-12: Full Angel with one shared and one raw serial port

Note that there can be other devices below `devclnt.c` and/or `devraw.c` (eg. PID Ethernet).

`devappl.c` provides a SWI veneer to permit the rest of the framework to reside either in an Angel ROM or in an Angel library linked with the application.

`devshare.c` switches requests according to whether the underlying device is shared with Angel or is a raw device.

For shared devices, `devshare.c` also converts the blocking character-oriented application requests into asynchronous packet-oriented requests.

`devclnt.c` multiplexes in two directions: it accepts asynchronous requests from the application (DC_APPL) or from the Angel channels layer (DC_DBUG), and directs them to the appropriate device driver.

`devraw.c` simply switches requests between raw devices.

If the device driver has been built using the `serring.h` / `serpkt.c` / `serraw.c` framework, `serpkt.c` provides the framework for Angel/shared devices, based on the Receive and Transmit engines and the serializer. Similarly, `serraw.c` provides the framework for raw devices, supporting blocking requests for reading or writing a given number of characters.

`serring.h` defines the control structure used to glue the hardware-specific driver layer to `serpkt.c` and `serring.c`. The interface is based on two ring buffers. One is filled by the upper layers with characters requiring transmission, which are extracted by the lower layer and sent. The other is filled with received characters by the lower layer and emptied by the relevant upper layer.

The device-specific lower layer must provide the actual ring buffers and control structures required by `serpkt.c` and `serring.c`, since one of each is required for each physical device or port. It must implement an IRQ handler and/or poll handlers to fill the receive buffer and empty the transmit buffer, together with control routines and a `putchar()` function for kicking new transmissions into life.

Additional devices can choose whether or not to use `serring.h` / `serpkt.c` / `serraw.c` as appropriate.

8.10.2 Minimal Angel with one raw serial port

Figure 8-13: Minimal Angel with one raw serial port illustrates case B.

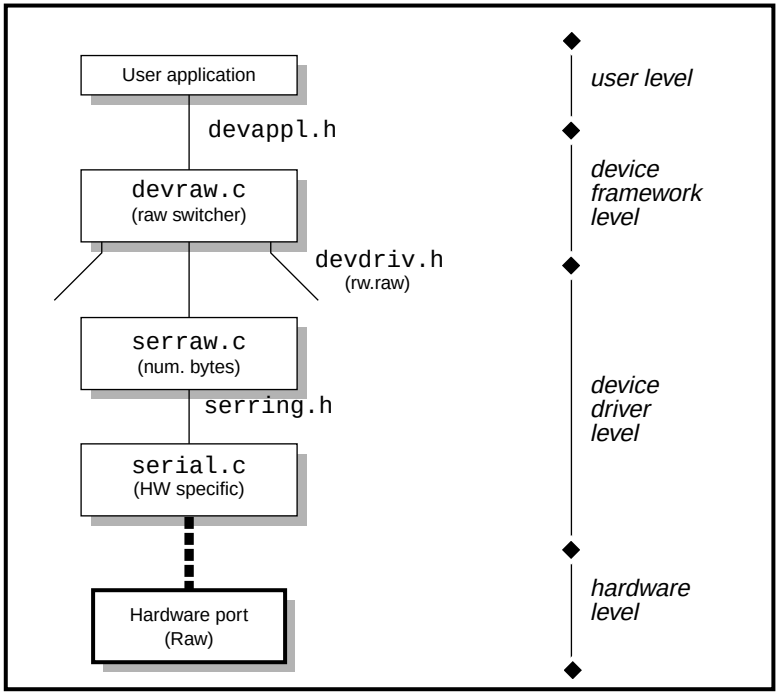


Figure 8-13: Minimal Angel with one raw serial port

In this case, `devappl.h` is simply a macro interface directly onto `devraw.c`. The remaining modules are identical to the right-hand flow in the full Angel case.

To move from case A to case C, substitute `serraw.c` for `serpkt.c`, and change the control structures appropriately.

In both examples, `devmisc.c` is not shown. This provides the core functionality for `..Yield()` calls.

8.10.3 User hooks for ADP communications in the host debugger

The ADP implementation for host debuggers provides two hooks for users:

- install a handler for incoming shared-device application packets
- install a handler for arbitrary asynchronous processing during target execution

The hooks are provided via `angsd/hostchan.h`. See also `angsd/devsw.h`.

Adp_Install_DC_Appl_Handler()

This routine installs a callback to handle packets which arrive from the application over a shared device (DC_APPL packets). A callback of NULL can be specified, in which case DC_APPL packets are simply dropped. The return value of the routine is the previously installed callback, if any.

When a packet arrives, it is passed to the callback along with the descriptor of the device. This descriptor can be used in the callback to send a reply, via `DevSW_Write()`.

Adp_Install_Async_Callback()

This routine installs a callback which will be called repeatedly whenever the debugger is waiting for a reply from the target. This includes the time when the target is executing. Up to eight async callbacks can be installed. Once installed, a callback cannot be removed.

Each callback is passed the descriptor of the active device (the one connected to the target) and the current time. The descriptor can be used if the callback needs to send a packet to the target. The time can be used to decide whether a periodic activity should take place.

Note *It is vitally important that any IO performed in an async callback is non-blocking. If a callback blocks, the whole debugger will be blocked. Similarly, a callback should not perform large amounts of time-consuming processing, otherwise the response time of the debugger will be adversely affected.*

Example

`angsd/unix/hostappl.c` is the default built-in processor for shared device communications between the application, and files or named pipes on the host.

In `hostappl_Init()`, `hostappl_async_cb()` is registered as an async callback. It is used to handle comms from the host to the target. Every time it is called, it attempts to read from `read_fd`, and if successful, it sends the data to the target via a call to `DevSW_Write()`. Note that when `read_fd` is opened in `hostappl_In()`, care is taken to ensure that it is marked non-blocking.

In `hostappl_Out()`, `hostappl_rx_handler()` is registered as the callback for DC_APPL packets. Whenever a packet arrives from the application, the contents are written out to `write_fd`.

Communications when target is idle

The hooks described above are necessary only for asynchronous processing while the target is executing. Extensions to the debugger which make use of new ADP channels can be implemented by following the pattern used throughout `angsd/ardi.c`:

```
int reason; register_debug_message_handler();
msgsend(<channel>, <format>, <reason> | HtoT, [<args>[...]]);
reason = <reason> | TtoH;
wait_for_debug_message(&reason, ...);
/* process response... */
```

This results in a synchronous transaction where a request is sent to the target and a response is awaited.

Note *Any async callbacks will get called during* `wait_for_debug_message()`*, whilst the host waits for the reply from the target.*

8.10.4 Error detection and reporting

The device driver is capable of either detecting errors in bad packets or of rejecting bad packets outright. Only correct packets are presented via callback. However, if the driver can detect errors or bad packets, a reception error should be presented to the next layer up via callback. This allows the channels layer to take appropriate action, such as requesting retransmission of a bad packet.

8.11 Fusion IP stack for Angel

8.11.1 How Angel, Fusion and the PID hardware fit together

The Ethernet interface for the PID card is provided by an Olicom EtherCom PCMCIA Ethernet card installed in either PCMCIA slot. The Olicom card uses an Intel i82595 Ethernet controller.

The UDP/IP stack is Pacific Software's Fusion product, ported to ARM and the Angel environment. The drivers for PCMCIA and the Ethernet card have been implemented by the Angel team, as has the Angel device driver to make the whole stack appear as an Angel device.

Figure 8-14: Angel, Fusion, and PID hardware shows how the components fit together.

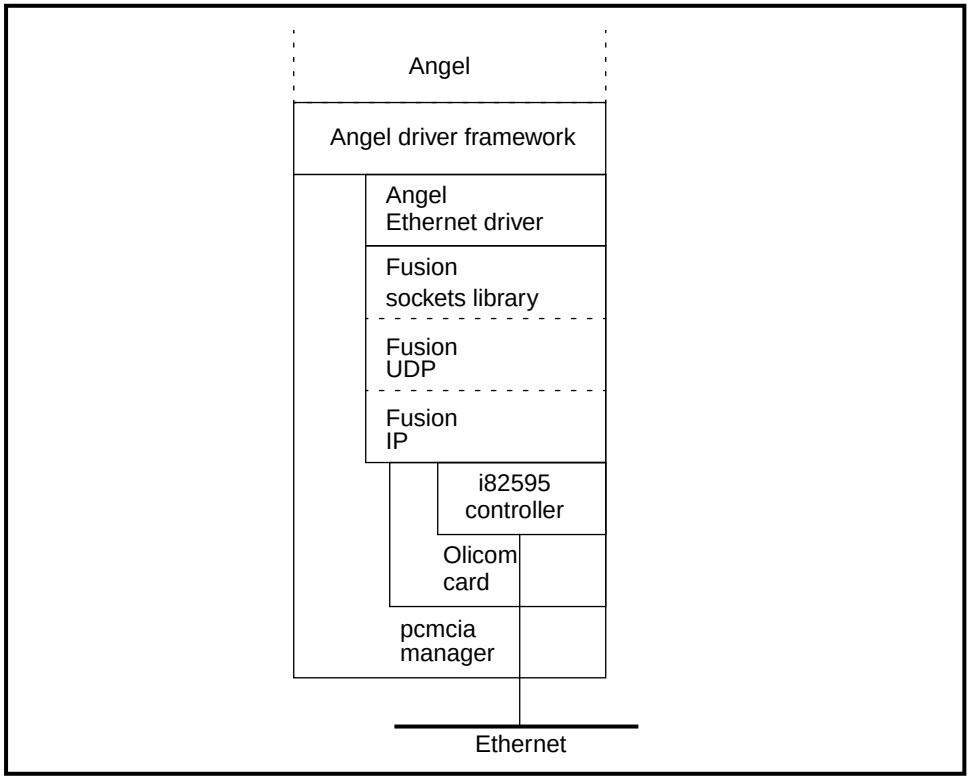


Figure 8-14: Angel, Fusion, and PID hardware

Directories and Files

`angel/ipstack/`
: Root of the IP stack

`angel/ipstack/ipconfig.c`
: IP address and netmask

`angel/ipstack/fusion/`
: Fusion source distribution

`angel/ipstack/fusion/arm/`
: ARM-specific Fusion build directory

`angel/ipstack/fusion/arm/config.h`
: ARM-specific Fusion configuration

`angel/ipstack/fusion/arm/os_arm.c`
: ARM-specific Fusion support funcs

`angel/ipstack/fusion/arm/Makefile`
: ARM-specific Fusion makefile

`angel/ipstack/fusion/incl/`
: Fusion header files

`angel/pid/82595.c`
: Low-level driver for i82595 controller

`angel/pid/ethernet.c`
: Angel driver for UDP over Ethernet

`angel/pid/olicom.c`
: Driver for Olicom EtherCom card

`angel/pid/pcmcia.c`
: Driver for PCMCIA controller

Initialization

The stack is initialized in the following sequence:

1. `devclnt.c:angel_InitialiseDevices()` calls:
2. `ethernet.c:ethernet_init()` which opens a socket.
3. `fusion:socket()` notices that the fusion stack has not been initialized.
4. Fusion stack initialization calls:
5. `olicom.c:olicom_init()` calls:
6. `pcmcia.c:pcmcia_setup()` detects Olicom card and calls:
7. `olicom.c:olicom_card_handler()` with a card insertion event, thus:
8. `olicom.c:read_card_params()` which registers `olicom_isr()` with `pcmcia.c`.
9. Fusion stack initialization calls:
10. `olicom.c:olicom_updown()` and, via `olicom_state()`:
11. `82595.c:i595_bringup()` to complete the initialization sequence.

Angel Ethernet device driver

After initialization, the Angel side of the driver is implemented as a polling device. At every call to `Angel_Yield()`, `angel_EthernetPoll()` is invoked, and non-blocking `recv()` calls are made to the Fusion stack to see if data is waiting on any of the sockets.

Outgoing packets are passed to the Fusion stack in a single step by calling `sendto()`.

Interrupt handling

The bottom of the Fusion stack is driven by interrupts from the Olicom card. Interrupts are handled in the following sequence:

1. `suppasm.s`:`angel_DeviceInterruptHandler()` calls the `GETSOURCE()` macro in `pid/target.s` to identify the PCMCIA controller as the source.
2. `pcmcia.c:angel_PCMCIAIntHandler()` establishes that it is an I/O interrupt and calls the routine registered during initialization.
3. `olicom.c:olicom_isr()` checks the interrupt, switches off interrupts from the Olicom card, and serializes `olcom_process()` to do the processing with all other interrupts enabled.
4. `olicom.c:olicom_process()` identifies the reason for the interrupt and passes it as an event to `olicom_state()`.
5. `olicom.c:olicom_state()` calls an appropriate routine in `82595.c` to handle packet reception and transmission.
6. `82595.c` routines control the i82595 chip and transfer packets in both directions between Fusion buffers and the chip. Calls are made to Fusion functions as appropriate.
7. `olicom.c:olicom_process()` checks to see whether all interrupt events have been serviced. If so, Olicom interrupts are re-enabled. If not, `olicom_process()` re-queues itself and then exits in case another device is waiting for the serializer lock.

Additionally, the Fusion stack can make calls to `olicom_start()` (to queue a new packet for transmission), `olicom_ioctl()`, and `olicom_updown()` in response to socket calls from the Angel Ethernet driver or as a result of packet processing.

Port numbering scheme

When a debugger is connected to an Angel target via UDP, two connections are used to two ports on the target. One carries DC_DBUG (Angel channels) packets and the other carries DC_APPL (application) packets.

To avoid having two reserved port numbers at the target, a single control port is opened at a well-known (hard-wired) address, and the two comms ports are dynamically allocated. When a host wishes to communicate with Angel, it sends a request to the well-known port, and the target replies with a message containing the dynamic port numbers allocated for the DC_DBUG and DC_APPL channels.

The target grabs the IP address and port of the host from the initial port request message, and uses them to send all subsequent replies to the host. The host uses `recvfrom()` to receive packets together with the details of the target's IP address and port number, and can then decide whether the packet is from the DC_DBUG or DC_APPL port.

8.12 Serialization, Stacks and Modes

This section gives an example of processing a simple packet from start to finish, and describes in more detail how serialization affects the receipt of data via interrupt.

Figure 8-15: Serialization shows the application running, when an interrupt request (`IRQ`) is processed. `SerialiseTask` is called (`S`). From point `S` execution is in SVC mode. The rest of the packet is then polled in, and enough of it interpreted to know which callback it is intended for. `QueueCallback` is then called (`Q`) which queues processing this packet.

Finally the packet assembly code returns, which is intercepted by `NextTask` code (`N`), which executes the queued callback. When this callback has finished, it returns. This is again intercepted by `NextTask` and then execution finally returns to the application where it left off for the initial IRQ.

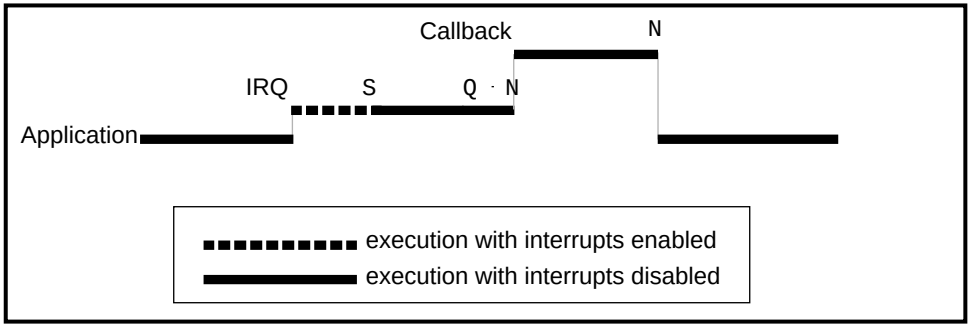


Figure 8-15: Serialization

The rest of this section describes in detail how this works, including more complex cases such as nested packets, hitting breakpoints, and so on.

8.12.1 Serialization

The lock being acquired is the mutually exclusive right to assemble a packet or to do initial processing when a breakpoint (undefined instruction) or special ReportException SWI (similar to a breakpoint) happens. It also allows mutually exclusive access to the Angel SVC mode stack, which is always empty when the lock is acquired and which must be empty when the lock is released. This ensures that it is never necessary to have more than:

- one user mode Angel stack area, which may be used simultaneously by a number of callbacks
- one application stack (per thread)
- one Angel SVC mode stack for use by code which has acquired the lock
- small, temporary IRQ, FIQ and UND stacks

Note *See* `serlock.h` *and the implementation in* `serlock.c` *and* `serlasm.s` *for full details of the functions described in the following sections.*

SerialiseTask

`SerialiseTask` can be called from IRQ, FIQ, SVC, or UND mode, but never USR mode. The calling mode depends on why it is being called. For example:

IRQ mode	if being called due to an IRQ from an IRQ driven device
UND	if due to a breakpoint being hit
SVC	if due to a SWI needing to call SerialiseTask and so on

`SerialiseTask` is passed a pointer to a register block (the state of the CPU when interrupted), the function which is to be called when serialization is successful, and a word of data to be passed to that function.

`SerialiseTask` sets up a register block for the task to be started with the mutual exclusion lock, and then determines whether the lock is already taken. The serialization module must also keep track of the priority of the task executing at all times (see `QueueTask` for the priorities). This is possible because the serialization module knows the priority of any task it allows to execute.

- If the lock is *not* taken:
  `SerialiseTask` sets the lock and calls `QueueTask` with the register block for the interrupted task, using its former priority level. `SerialiseLTask` then sets up the registers to the values in `desired_regblock` with interrupts enabled and LR modified to ensure that execution returns to `NextTask` when the task that is given the lock finishes.
- If the lock *is* taken:
  `SerialiseTask` calls `QueueTask` with the register block of the new function wanting the lock, and with the priority of `AngelWantLock`. It then restarts execution using the register block for the interrupted task, which must be the one with the lock.

Task priorities

These priorities are shown from lowest to highest:

`IdleLoop`	
`AngelInitialisation`	
`Application`	The user's application
`ApplicationCallBack`	Callbacks for the user's application
`AngelCallBack`	Callbacks with Angel
`AngelWantLock`	Code needing the serialization lock

This function executes with interrupts disabled.

QueueCallback

This function is intended to be called by:

- device drivers
- breakpoint handlers
- SWI handlers
- or the `Angel_Yield` code, which is executing in SVC mode with the lock

It is used to queue a callback, specify its priority and also specify up to four arguments to that function. The callback does not execute immediately, but will start when all tasks of a higher priority have completed.

BlockApplication

This function is called in order to allow or disallow any application tasks being executed. While suspended they remain queued, but are not executed.

NextTask

This is not a function, in that it is not called directly. `NextTask` is executed by code which has been set running by either `SerialiseTask` or `NextTask` itself. This is done by setting up LR to point to `NextTask` before a task is run.

When the `NextTask` code is executed, the processor will either be in:

SVC mode for code with the lock

User mode for callback code

Interrupts are enabled when `NextTask` is entered.

The operation of NextTask can be summarized as follows:

1. `NextTask` must get into SVC mode and disable interrupts.
2. `NextTask` looks at the queue of requested tasks, and selects a task which has not been blocked. Of these it chooses the one with the highest priority, and if there are several with the same priority it chooses the one which was requested first.
3. `NextTask` removes the task from the queue of tasks and sets up the registers as specified in the register block for that task. `NextTask` sets up LR so that `NextTask` is executed once again when this task returns.

If the task to be executed was queued with the flag set, `NextTask` must set up a suitable environment for it, as follows:

Application callback: For an application callback, the environment must be set by:

- searching for the logical parent of the callback (the application task) in the list of queued tasks, and
- inheriting sl and sp from the parent

The task executes in user mode with interrupts enabled.

Angel CallBack: For an Angel callback, the environment is set by searching the queue of tasks for Angel callbacks and noting the lowest value of sp amongst these tasks. Sp is set by a fixed amount below this value and sl for the queued task is reset to this new value. Sl for the new callback is set to the old value of the selected queued task.

The task executes in user mode with interrupts enabled.

AngelWantLock: If the task being started has priority `AngelWantLock`, it is given the empty Angel SVC stack, and executes with interrupts enabled in SVC mode.

Note that the Application is treated just like any other task in that it is queued up and resumed. The differences between this and non-application tasks are:

- it has a lower priority than all Angel tasks except initialization and the idle loop
- it can be blocked

The idle loop: If there are no unblocked tasks to execute, interrupts should be enabled, and an idle loop entered. The current task is now an idle loop. When the last task executing was an idle loop, the next request to queue a task will be from `SerialiseTask` to queue the idle loop itself. This request is ignored. The idle loop repeatedly calls `Angel_Yield` and executes with interrupts enabled.

```
while (1) { Angel_Yield(); }
```

AccessApplicationRegBlock

This function returns a pointer to the register block for the Application task in the queue. Angel code uses this to read and modify the application's registers, when requested to do so by the debugger.

NULL is returned if the Application is not blocked (it is not legal to access the Application's registers while it is still running— it must have been stopped first).

Note that this only covers those registers saved in the Application register block. It does not provide a way for the debugger to modify banked registers being used by the Device Driver Architecture. Attempting to modify such registers while Angel is executing is not advisable. However, the banked registers are readable. This is managed in `debugos.c`.

FlushApplicationCallbacks

This function removes from the task queue any tasks with priority `ApplCallBack`. This needs to be done only if a new application is loaded or the old one is reloaded. In either case, any packets waiting to be processed by the old application are clearly no longer relevant and should be discarded.

8.12.2 Processor modes and stacks

Angel makes use of SWIs and Undefined Instructions as well dealing with IRQ and FIQ driven devices. It also has to deal with packet processing code, callback code and application code. However, the serialization mechanism described in ***8.12.1 Serialization*** on page 8-38 ensures that only one task ever executes with the lock. Therefore, all tasks which execute with the lock may share a single stack, on the basis that:

- it is always empty when a task starts
- once that task has finished (ie. it returns), all information which was on that stack is lost

This mechanism allows very efficient usage of stack space, because only one stack block is required for all packet assembly and initial breakpoint-processing code.

The application itself uses a different stack, and executes in either User or SVC mode. Application callbacks (ie. callbacks due to application requests to read or write from devices under control of the Device Driver Architecture) also execute in User mode and with the application stack.

8.12.3 Overview of Angel stacks for each mode

The following stacks within Angel are simple stacks exclusively used by one task. This is ensured by disabling interrupts in these modes:

- IRQ stack
- FIQ stack (if used)
- UND stack

The Angel SVC stack is also only ever used by one task at once, and this is ensured by the serializer. If the application uses the SVC stack, Angel ensures that this is kept separate from the Angel SVC stack.

The USR mode stack is also split into two cases, as the Application's stack and Angel's stack are kept entirely separate.

The Application stack

The Application stack is a simple single block. The application is assumed to be a single threaded application, with the single exception of Application Device Callbacks. Angel allocates these a "temporary" stack, below the current stack pointer for the Application, but in the same stack block. This is acceptable because the Application cannot resume until the Application callback has completed. Note that the application can run in either USR or SVC mode. However, if it runs in USR mode, it should only enter SVC mode using `Angel_EnterSVC`. If it is running in SVC mode, it should not drop down to USR mode without saving away the SVC mode context if it needs to.

The Angel USR mode stack is similar to the Application USR mode stack in that it is a single block. However, its management is considerably more complex.

The Angel USR mode stack

The Angel USR mode stack is used only by tasks of priority AngelCallback (ie. Angel callback tasks).

The following set of rules are used to set up the Angel USR mode stack when starting up a new task of priority AngelCallback:

Is there already a task of priority AngelCallback?

No	sp = AngelStack sl = AngelStackLimit
Yes	Search through all such tasks and find the task with the lowest sp. Modify sl for that task to be sp-*val*, where *val* is a carefully chosen constant. The new task is then set up with: sp = other task's modified sl sl = AngelStackLimit

Note that this limits the stack available to the old task, but it should be possible to choose a value of val such that no stack overflow occurs. This value needs to be chosen with care to ensure that no stack overflow happens. Also, RAM may be scarce, in which case large amounts of stack space may not be available.

Stack overflow

Overflow on any of the stacks apart from the Application USR stack is considered a fatal error, and Angel makes no attempt to report back to the debugger. Instead, it simply loops forever at a point which allows investigation to show what has happened.

However, Application stack overflow is reported to the debugger, so that the application programmer can allow the application more stack.

If the application needs to execute in SVC mode for a while (for example, to access protected IO space), it may do so if it adheres to the rules described in ***Get into SVC mode*** on page 8-47.

IRQ/FIQ interrupt handling

On entry to IRQ and FIQ, further interrupts are disabled. They must not be re-enabled before `SerialiseTask` is called, or if the interrupt completes (if it is an interrupt for a peripheral which is not a packet receiver device).

There are some circumstances in which it would be safe to leave FIQs enabled. This is discussed in ***8.12.8 Reducing FIQ latency*** on page 8-51.

The IRQ/FIQ code may use the IRQ/FIQ stack, but before calling `SerialiseTask`, everything on the stack should no longer be required, and the stack pointer returned to where it was when the interrupt happened (the stack must be left empty after use).

The interrupt handler saved the state of the processor at the time the interrupt handler was entered, because this data is required by `SerialiseTask`.

`SerialiseTask` is then called and the device driver task will execute at some point. For example, a serial device could then poll in the rest of the packet.

When a complete packet has arrived and been decoded, it can be passed to a distribution function which determines which packet-handling callback is called. However, it will not actually call the distribution function, but calls `QueueCallback`, specifying the priority to be that of a callback routine of the appropriate type (`AngelCallBack` or `ApplCallBack`).

The packet assembler and despatcher can return, which causes NextTask to execute, and the callback to process this packet executes in User mode, without the lock. When this has finished processing it can return, and at this point the packet indicated by the initial interrupt has been completely processed.

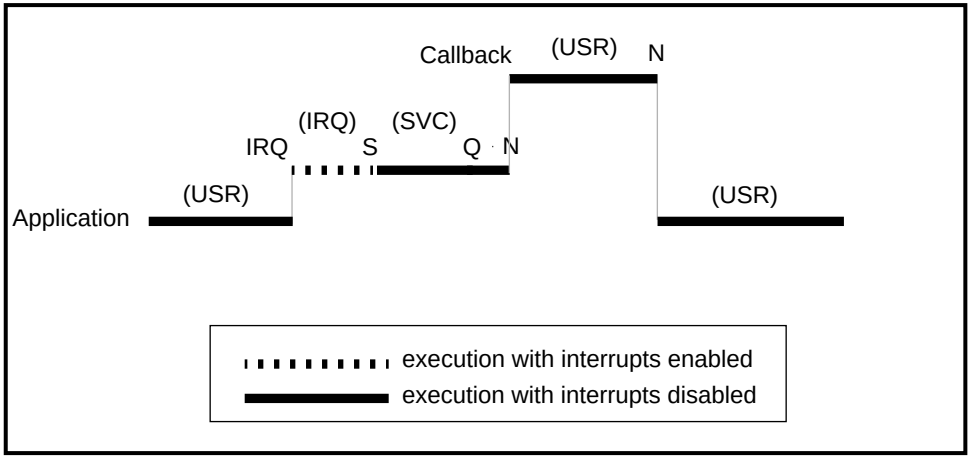


Figure 8-16: IRQ/FIQ interrupt handling

Undefined instruction (breakpoint)

A breakpoint can be considered another source of callback requests. In order to protect against other breakpoints happening while one breakpoint is being processed (in a multi-threaded system), or another packet arriving and being processed in the middle of processing a breakpoint, the serializer must be used by the breakpoint mechanism in a similar way to that used for an IRQ.

On entry to the undefined instruction handler, IRQs and FIQs are disabled until `SerialiseTask` is called. (In some circumstances FIQs need not be disabled; this is described in ***8.12.8 Reducing FIQ latency*** on page 8-51.)

The undefined instruction handler saves the state of the processor at the time the undefined instruction happened, because this data is required by `SerialiseTask`. The undefined instruction handler may use the UND mode stack, provided that it leaves the stack empty when it calls `SerialiseTask`.

`SerialiseTask` is called, and at some point the process breakpoint code executes. This calls `QueueCallback`, specifying Angel callback function to process the breakpoint. (Minimizing the time the lock is held enables other packets to be processed quickly.)

The User mode callback then executes, and sends a packet to the host, indicating that a breakpoint has been hit. It also calls `BlockApplication` to ensure that the application does not restart until an execute request has been received.

To wait for an acknowledgement of the packet sent to the host, register a temporary handler with the channel manager and then finish. To do this safely, the register call should be made before the packet is sent. At some later point, this temporary callback will in turn be called and the acknowledgement can be checked. Finally this callback can also finish.

At this point, `NextTask` has only the Application task (and possibly an application callback task) on its queue, but both of these will be blocked. Therefore, `NextTask` enables interrupts and sits in an idle loop; the system is waiting for more requests from the debugger at this point.

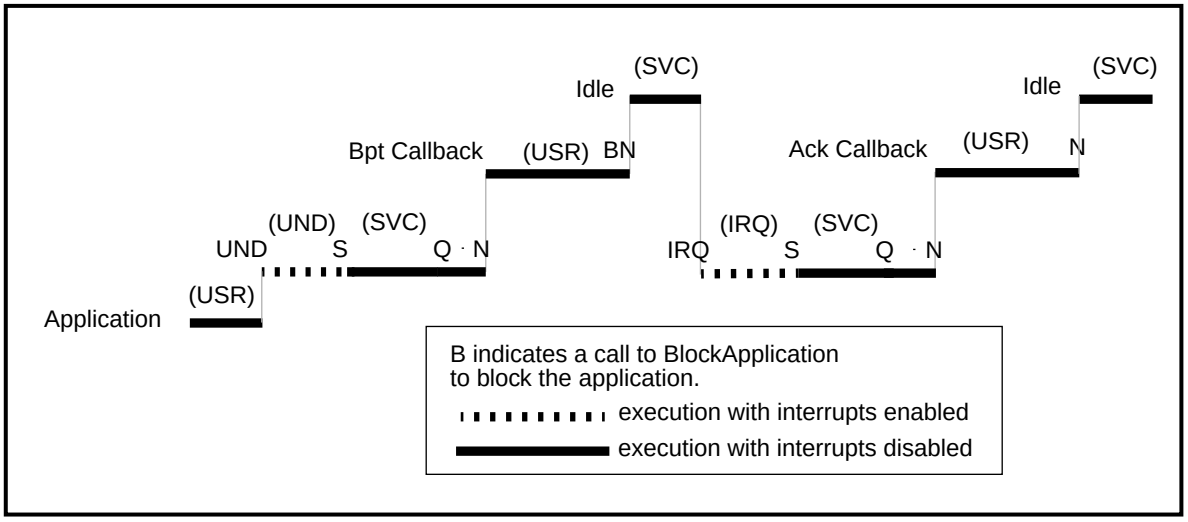


Figure 8-17: Breakpoint callback (1)

As an alternative, the breakpoint callback may choose to perform a blocking read to receive the acknowledgement. This is translated by the channels layer, which registers an internal handler and enters a yield loop. The acknowledgement is then processed as a nested callback which unblocks the read occurring in the original callback.

Note that this is permitted by the serializer architecture, and that other interrupts and callbacks are free to happen during the blocking read. Only the thread and the channel involved in the “blocking read” are actually blocked.

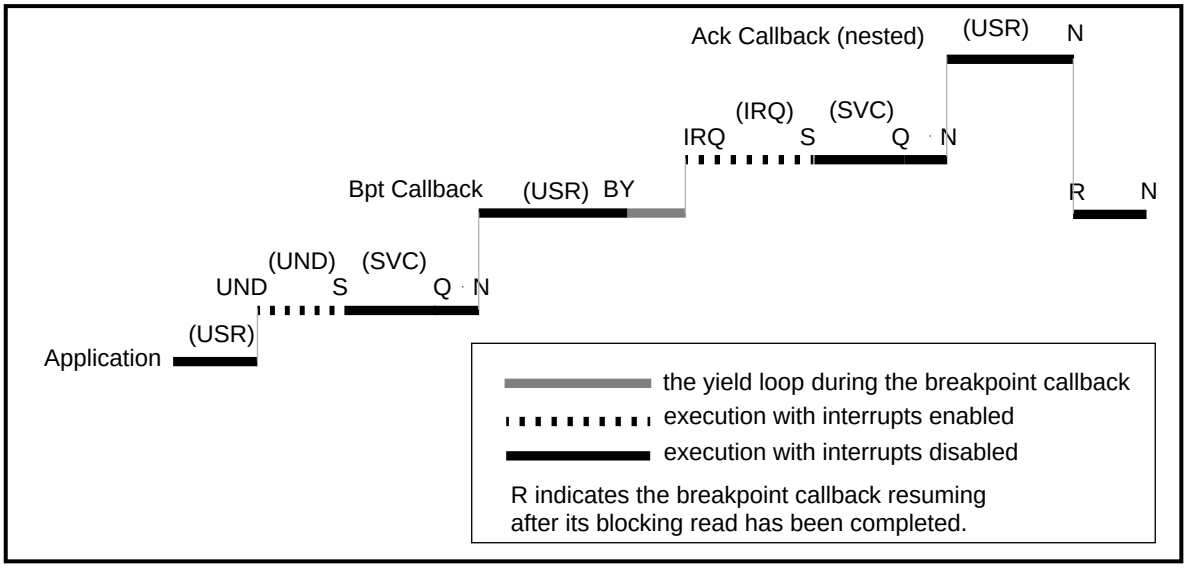


Figure 8-18: Breakpoint callback (2)

Software interrupts (SWIs)

Angel uses a single SWI, but r0 is used as a reasoncode to determine what that SWI should do. These reasoncodes fall into the following categories:

- ReportException, which is used by the semihosting support code as a way to report an exception to the debugger. It can be considered as a breakpoint which starts off in SVC mode rather than UND mode.
- Calling veneer only, eg. semihosting support and device driver access
- Get into SVC mode

These are considered in detail in turn below.

ReportException: This is effectively a breakpoint, but one which is implemented via a SWI rather than via an undefined instruction, as distinguished from a normal breakpoint. It also has to pass a parameter to the debugger, showing the type of the exception that occurred.

Because `ReportException` is used only by the application (while executing part of the semihosted C Library), and the application is executing only when there are no tasks with the lock running (or indeed any callbacks running), it is clear that the Angel SVC stack used by tasks with the lock is empty, and not in use at this point. Furthermore, as interrupts must be kept disabled until `SerialiseTask` is called, at which point any stack used must be empty, no task can use the SVC stack when the `ReportException` code would want to use it.

Therefore, the ReportException code may make use of the Angel SVC stack as temporary workspace with the standard restrictions which `SerialiseTask` imposes.

Calling veneer SWIs: In this case, all that is required is that a SWI is used to get into Angel. The code which is called indirectly is executed using the application's stack. Like `ReportException` it is only called from the application, and therefore the argument that the Angel SVC stack can be used as temporary workspace while interrupts are disabled still applies.

What this does mean is that the Angel SVC stack cannot be used to save the return address into the application while the indirectly-called function executes. This data must be held on the application's own stack.

Get into SVC mode: This SWI:

- switches into SVC mode
- disables IRQs and FIQs. There are some circumstances in which FIQs need not be disabled; these are described in ***8.12.8 Reducing FIQ latency*** on page 8-51.
- switches the caller's stack pointer into the SVC stack pointer (to keep the code APCS-compliant)
- leaves the caller's CPSR in SPSR_SVC

The caller must not re-enable interrupts before switching back into user mode.

When the caller no longer needs SVC privilege, `Angel_ToUSR` should be called. For convenience, the address of this function is returned in r0 by `Angel_EnterSVC`, which is a C veneer on this SWI.

8.12.4 Continuing execution after a breakpoint

When the application has hit a breakpoint, and is therefore not running (see ***Undefined instruction (breakpoint)*** on page 8-44), the Application task and possibly some application callbacks are queued, but all are marked as blocked.

The debugger makes the application continue execution by sending an execute packet to Angel. This is processed in the same way as any other packet; see ***IRQ/FIQ interrupt handling*** on page 8-43 for details.

The only difference is that, as well as sending a reply to the execute packet, `BlockApplication` is called to unblock the application. When the execute packet processing callback finishes (and any other outstanding packet processing requests or callbacks are also completed), `NextTask` automatically restarts the application, because it is on the task queue and is unblocked.

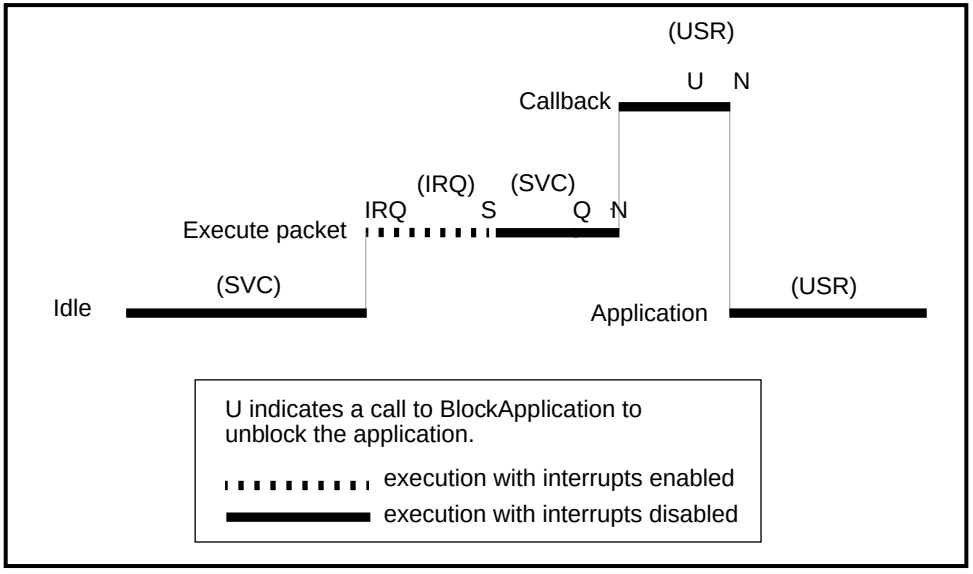


Figure 8-19: Continuing after a breakpoint

8.12.5 User interrupt requests

When the application is running there are no tasks queued.

The debugger makes the application halt execution by sending an interrupt execution packet to Angel. This is processed in the same way as any other packet; see ***IRQ/FIQ interrupt handling*** on page 8-43 for details.

The only difference is that, as well as sending a reply to the interrupt execution packet, `BlockApplication` is called to block the application. When the interrupt execution packet processing callback finishes (and any other outstanding packet-processing requests or callbacks are also completed), `NextTask` automatically sees that the application is now blocked, because it is on the task queue and is blocked. It therefore sits in an idle loop awaiting further requests from the debugger.

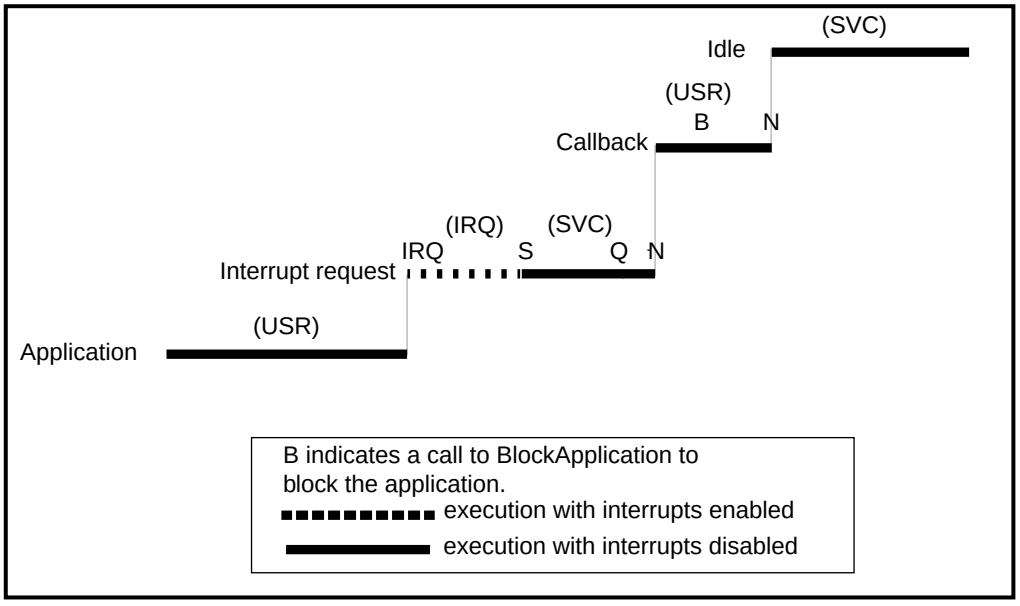


Figure 8-20: User interrupt requests

8.12.6 The yield function: Angel_Yield

A yield function for polled devices can be called both by the application and by Angel, while sitting and waiting for something to arrive on a polled device, or within cpu-bound loops (eg. the Idle Loop).

`Angel_Yield` can be considered as another form of interrupt: it is just one which you explicitly allow to happen rather than it being forced to happen.

Therefore, the same serialization mechanism can be used for `Angel_Yield` as is used for IRQ interrupts. As with an IRQ it can be called from either USR or SVC mode, and so must be able to return cleanly to either mode.

`Angel_Yield` may use whatever stack it was called with, but it must appear to be an APCS-conformant function from its caller's point of view, and so must leave the appropriate registers unaltered and the stack as it was when it returns. It will need to get into SVC mode if not called in that mode, so that it can disable interrupts.

Once `Angel_Yield` has acquired the lock, it should call each of the registered polling routines, and each can then queue a callback if there is data ready to be processed. It should go around this loop once.

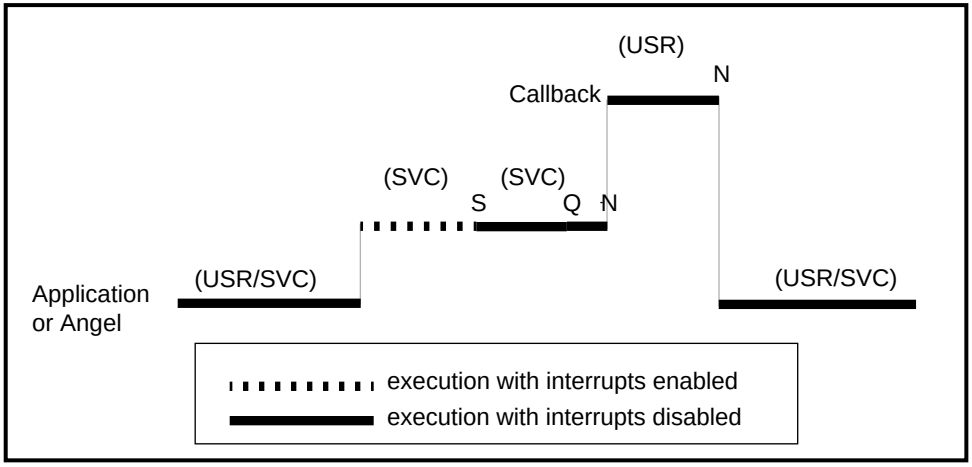


Figure 8-21: Angel_Yield

Note that sometimes the Yield code may not find any work which needs to be done, in which case no callback will be queued.

8.12.7 Allocation and deallocation of register block structures

Two shared register blocks: angel_MutexSharedTempRegBlocks

There are two statically allocated, but global, register blocks, which are used to hold:

- the registers at the time of an interrupt/SWI/undefined instruction
- the registers required by a new task to be executed

These are used to pass this data into `SerialiseTask`. `SerialiseTask` then ensures that any data which is needed in the future is copied out of them.

It is possible to share these two structures because all of the above handlers *must* keep interrupts (IRQ and FIQ) disabled until `SerialiseTask` is called; these structures are protected by mutual exclusion.

A pool of centrally managed register blocks

There is a configurable pool of register blocks, which are private to the serializer module.

Whenever a task is queued, a free register block is taken from the pool, and the data from the temporary register block is copied into the register block chosen from the pool.

When a task is unqueued by `NextTask`, the register block from the pool for that task is marked as free.

If these is no free register block when queuing a task, this is a fatal error.

8.12.8 Reducing FIQ latency

In practice, many systems could be less restrictive with FIQs than stated here, enabling them in some places where they have been disabled by default. The following are defined and are modifiable to suit your system.

The `#define FIQ_SAFETYLEVEL` can be used to reduce FIQ latency in the following ways:

- In the IRQ-handling code which runs in IRQ mode, FIQs can be kept enabled if:

  ```
  FIQ_SAFETYLEVEL >= FIQ_NeverUsesSerialiser
  ```

- Within the SWI handler, Yield function, and also within code which has entered SVC mode by using the “Get into SVC mode SWI”, FIQ can be kept enabled if:

  ```
  FIQ_SAFETYLEVEL >= FIQ_NeverUsesSerialiser_DoesNotReschedule
  ```

- Within the undefined instruction (breakpoint) handler, code which runs in UND mode can keep FIQs enabled if:

  ```
  FIQ_SAFETYLEVEL >= FIQ_NeverUsesSerialiser_DoesNotReschedule_HasNoBreakpoints
  ```

where:

`FIQ_CannotBeOptimised`
: means that FIQ is always disabled whenever IRQ is disabled.

`FIQ_NeverUsesSerialiser`
: means that FIQs never need to use the serializer to gain the lock. For example, FIQs might be used by the serial device (as in the ARM60 PIE card default configuration).

`FIQ_NeverUsesSerialiser_DoesNotReschedule`
: adds to the limitation of `FIQ_NeverUsesSerialiser`, in that FIQs must not be used as a means for a third-party operating system to reschedule tasks.

`FIQ_NeverUsesSerialiser_DoesNotReschedule_HasNoBreakpoints`
: adds to the limitations of `FIQ_NeverUsesSerialiser_DoesNotReschedule`, in that breakpoints must never be set in code which is executed in FIQ code.

The default is the “safest” setting:

```
#define FIQ_SAFETYLEVEL FIQ_CannotBeOptimised
```

8.12.9 Checking for “impossible” cases

To speed up debugging, you need to include runtime assertion code to check that the state of Angel is as expected. Such assertions should be within the protection of:

```
#if ASSERTIONS_ENABLED
```

so that you can disable them in a final version if required.

Such assertions have been made wherever possible; for example, when it is assumed that a stack is empty, there are no items in a queue, and so on.

8.12.10 The device driver's view

Both Angel callbacks and Application callbacks run in USR mode, without the lock. Angel callbacks can make direct calls to functions described in `devclnt.h`, but Application callbacks have to do so indirectly through the SWI veneer.

Two pseudo-functions (in `serlock.h`) are needed to enter SVC mode and disable IRQ and FIQs, and also to keep the caller's stack accessible. These two functions are used to get mutually exclusive supervisor access to data structures and IO space:

```
void Angel_EnterSVC(void)
void Angel_ExitToUSR(void)
```

The device driver function called by Angel or the Application does almost no work itself; it just asks for the request to happen, or at most only does tasks which cannot block and which do not require a period of waiting.

Then either an interrupt-driven or polled handler does the bulk of the processing.

The following sections show how this works for fully interrupt-driven, partially-polled and fully-polled devices.

Note that the polling routine is not unregistered; it is a permanently installed read poller.

Fully IRQ-driven write request

In the function called by `devclnt.c` the following happens:

1. `Angel_EnterSVC()`
2. Note the user callback
3. Send first byte
4. Enable "write byte complete" interrupts on device
5. `Angel_ExitToUSR()`

Every time the device has written a byte, an interrupt grabs the lock. The driver then writes another byte. When the entire packet has been written, the callback is queued and the device has its 'write byte complete' interrupts disabled.

Fully IRQ-driven read request

In the function called by `devclnt.c` the following happens:

1. `Angel_EnterSVC()`
2. Note the user callback
3. Enable "byte read" interrupts on device
4. `Angel_ExitToUSR()`

Every time the device has read a byte, an interrupt happens and grabs the lock. The driver then puts this byte into the buffer. When the entire packet has been read, the callback is queued. Note that read interrupts for this device are left enabled.

Half-IRQ, half-polled read request (no equivalent for write requests)

The function called by `devclnt.c` to set this up is identical to the fully IRQ-driven function.

On receiving a read interrupt, the driver:

1. gets the first character which arrived and caused an interrupt
2. disables read interrupts from happening from before getting the lock (before re-enabling IRQ and FIQ)
3. gets the lock (by calling `SerialiseTask`) until the packet is complete
4. calls `QueueCallback` for the packet
5. return, giving back the lock

Polled write

In the function called by `devclnt.c` the following happens:

1. `Angel_EnterSVC()`
2. note the user callback
3. send first byte
4. register polling routine to be called by `Angel_Yield`
5. `Angel_ExitToUSR()`
6. `Angel_Yield()`; this gives an immediate chance for the write to go ahead

There is no interrupt routine, but there is a polling routine which was registered to be called by `Angel_Yield`. This does the following:

```
Ask if the device ready to write another byte.
If not, return
Repeat {
   Write another byte
   If that was the last byte in the packet, break
   Wait until device is ready to write another byte
}
Queue Callback
Unregister polling routine
Return
```

Polled read

In function called by `devclnt.c` the following happens:

1. `Angel_EnterSVC()`
2. note the user callback
3. register polling routine to be called by `Angel_Yield`
4. `Angel_ExitToUSR()`
5. `Angel_Yield()` gives an immediate chance for the read to happen

There is no interrupt routine, but there is a polling routine which was registered to be called by `Angel_Yield`. This does the following:

```
Has the device read a byte yet?
If not then return
Repeat {
   Put the byte in the buffer
   If that was the last byte in the packet, then break
   for (i=0; i<chosen_value; i++) {
          Has device read another byte ?
          If yes, then continue from top of repeat loop
   }
   return - out of time and not got a whole packet
}
Queue Callback
Return
```

9 ARMulator

This chapter describes the ARMulator, ARM's instruction set simulator.

9.1 About the ARMulator

The ARMulator is a collection of programs that emulate the instruction sets and architecture of various ARM processors. It provides an environment for the development of ARM-targeted software on the supported workstation and PC host systems.

The ARMulator is instruction-accurate: it models the instruction set without regard to the precise timing characteristics of the processor. As a result, it is well suited to software development and benchmarking of ARM-targeted software, though its performance is slower than real hardware. ARMulator also supports a full ANSI C library to allow complete C programs to run on the emulated system.

ARMulator is transparently connected to the ARM symbolic debugger 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 or C++ which interface to the ARMulator's external interface.

Note *The ARMulator's interfaces have been extended at Version 2.1, and are now different from the Version 2.0 interfaces.*

For additional information about the ARMulator, refer to the *Software Development Toolkit User Guide (ARM DUI 0040).*

9.2 Modelling an ARM-based System

An ARMulator environment consists of five parts:

- Remote Debug Interface
  the interface between the ARMulator and its host debugger
- ARM Core Model
  the model of the ARM processor itself (eg. ARM6, ARM710, StrongARM)
- Memory Model
  the model of the memory system outside the ARM, typically just RAM. ARMulator models are more dynamic than mapfiles used in `armsd`, allowing, for example, memory-mapped I/O.
- Coprocessor Models
  these model ARM coprocessors directly
- Operating System or Debug Monitor model
  a virtual interface between the host and the modelled ARM

In addition there may be “basic” models. These do not form part of the ARMulator, but may exist around it providing extra functionality.

The Remote Debug Interface and ARM Core models are built into ARMulator, but you can add basic, memory, coprocessor, and operating system models.

9.2.1 Model stubs

Basic models, memory models, coprocessor models, and operating system models all attach to the ARMulator through a small stub. This stub consists of an initialization function and a textual name for the model, which ARMulator uses to locate it. Some example implementations exist to help with implementing new models. Any number of models can be attached to an ARMulator without the need to modify existing models—which model is used can be set when the ARMulator is run without need for recompilation.

At startup, the ARMulator locates the model, then calls the initialization function, passing in a list of pointers that the model should fill in with implementation functions. The model should also register an `ExitUpcall` (see ***ExitUpcall*** on page 9-17) during initialization, to free any state it may set up.

9.2.2 Configuration

ARMulator provides a method of configuring models called `ToolConf`. The `ToolConf` is a simple database of tags and values, which are read by ARMulator during initialization from a configuration file (`armul.cnf`).

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 of type, called `toolconf`.

The most useful functions are:

```
const char *ToolConf_Lookup(toolconf db, const char *tag)
```

This looks up `tag` in the configuration database db, returning the value associated with it, or NULL if the tag is not used in the database. (The tag is case-independent.)

```
int ToolConf_Cmp(const char *value, const char *reference)
```

This function compares a looked-up tag with an expected value, ignoring case. Returns TRUE if value matches reference.

Although documented in `toolconf.h`, the format of the `armul.cnf` file is not fixed, and is not guaranteed to remain the same in future releases.

9.2.3 ARMul_State

The `ARMul_State` is an opaque datatype which is a handle onto the ARMulator for the access functions. See ***9.7 Accessing the ARMulator's State*** on page 9-15.

9.3 Basic Model Interface

The simplest model interface is the Basic interface. This provides a mechanism for calling a user-supplied function during initialization. This function can then install callbacks, and so on, to add functionality.

9.3.1 Initialization function

The basic model exports a function which is called during initialization. The ARMulator calls the function only if an entry is found for it in the configuration file.

```
ARMul_Error init(ARMul_State *state,
       toolconf config)
```

This function returns zero (equivalent to `ARMulErr_NoError`) to indicate success, or an `ARMul_Error` value (from those defined in `errors.h`), via `ARMul_RaiseError`. Returning an error indicates a failure to initialize, and the ARMulator reports this failure back to the debugger.

9.4 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, passed in a flags word to determine which type of access is being performed. This flags word consists of a bitfield corresponding to the signals on the outside of the ARM.

As there are many core processor types, there are many variants of the memory interface. The memory initialization function is told which variant it should provide. A model must refuse to initialize for a variant it does not understand.

9.4.1 Initialization and other functions

The memory model is initialized through the initialization function declared in its stub. This is passed the `ARMul_State`, a pointer to the table of functions to be filled in, the variant of the memory model being requested and the model's `toolconf`.

```
ARMul_Error init(ARMul_State *state,
      ARMul_MemInterface *interf,
      ARMul_MemType variant,
      toolconf your_config)
```

The function returns zero to indicate success, or an error number.

The initialization should set the *handle* for the model, by assigning to `interf->handle`. This handle, usually a pointer to the state representing this instantiation of the model, is passed to all the access functions called by the ARMulator. (This single handle replaces the following four pointers used in previous ARMulators: `MemDataPtr`, `MemInPtr`, `MemOutPtr`, and `MemSparePtr`.)

This function should also be used:

- to register any upcalls (in particular an exit upcall if necessary)
- to announce itself to the user (using `ARMul_PrettyPrint`; see ***9.8.1 Input/output functions*** on page 9-26)
- to attach any associated coprocessor models (a memory-management unit, for example) and to set up its state

Which functions should be written into the function table depends on which variant is being requested. The table consists of some functions common to all models, and a union of functions which are specific to each type.

The common functions are:

```
unsigned long read_clock(void *handle)
```

`read_clock` is called whenever the ARMulator needs to know the elapsed time. The value returned should be the number of emulated microseconds since the model was initialized. A model can supply NULL as this function, if it does not support this functionality.

```
typedef struct {
       unsigned long NumNcycles,
       NumScycles,
       NumIcycles,
       NumCcycles,
       NumFcycles;
       } ARMul_Cycles;
const ARMul_Cycles *read_cycles(void *handle)
```

A model may keep count of the accesses made to it from the ARMulator. These counters are used to supply the `$statistics` variable inside armsd or the ARM Debugger for Windows.

`read_cycles` is called each time the counters are read.

9.4.2 Basic memory interface

There are three kinds of basic memory interface, but all three use the same function interface to the core:

`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 (which may be sequential). This may indicate to a memory model that *most* accesses will be halfword-instruction-sequential rather than the usual word-instruction-sequential.

Memory models which 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.

```
int mem_access(void *handle,
       ARMword address,
       ARMword *data,
       ARMul_acc access_type)
```

On each ARM core cycle, `mem_access` is called, where:

`handle`	is the value assigned to `interf->handle` in the initialization function
`address`	is the value on the ARM's address bus
`data`	is a pointer to the data to be stored (write), or a pointer to where a value should be written (read), or may be NULL in the case of a non-memory (idle) cycle
`access_type`	determines the cycle type

The function returns one of the following codes:

`1`	indicates successful completion of the cycle
`0`	indicates that the processor should busy-wait and try the access again next cycle
`-1`	returns an abort
For reads	the function should write the value to be read to the word pointed to by `data`; for a byte load it should write the byte value, for a halfword load the halfword value and so on. The model does not need to worry about the alignment of the address passed to it, because this is handled by the ARMulator. However, it does need to present the bytes of the word in the correct order for the endianness of the processor. This can be ascertained by using either a `ConfigChange` upcall or `ARMul_SetConfig` (see ***9.7 Accessing the ARMulator's State*** on page 9-15). `armdefs.h` provides a flag variable/macro, `HostEndian`, which is TRUE if ARMulator is running on a big-endian machine. See the `armflat.c` sample file for how to handle endianness.
For writes	`data` points to the datum to be stored. This value may need to be shortened for a byte or halfword store, however. As with loads, endianness must be handled correctly.

Note *In previous versions of the ARMulator, memory models used* `ARMul_DATAABORT` *and* `ARMul_PREFETCHABORT` *macros to return aborts. This is not necessary with the current release.*

Access type

The `access_type` encodes the type of cycle. On some processors (for example, cached processors) the signals will not be valid. The macros for determining this are:

`acc_MREQ(acc)`	choose between memory and non-memory accesses.
`acc_WRITE(acc)` `acc_READ(acc)`	for memory cycles, determine whether this is a read or write cycle (not `acc_READ` implies `acc_WRITE`, and vice versa).
`acc_SEQ(acc)`	for a memory cycle, true if the address is the same as, or sequentially follows from the address of the preceding cycle. For a non-memory cycle distinguishes between coprocessor (`acc_SEQ`) and idle (not `acc_SEQ`) cycles.
`acc_OPC(acc)`	for memory cycles, 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)`	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.

The **nTRANS** signal from the processor is not passed to the memory interface. As this signal changes infrequently and may well not be used by a memory model, a model should use the `TransChange` upcall (see ***TransChangeUpcall*** on page 9-18) to track **nTRANS**.

`unsigned long get_cycle_length(void *handle)`

`get_cycle_length` may be called by the ARMulator to return the length of a single cycle, in units of one tenth of a nanosecond. For example, it would return 300 for a 33.3MHz clock.

Cached versions

Three variants of this interface exist for cached processors (such as ARM610, ARM710):

`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 just a Memory cycle, with `address` incremented from the previous access.

Cached processors do not export the processor mode and **nTRANS** signals.

However, models can still register `ModeChange` and `TransChange` upcalls (and would still be called when the processor core changes these signals), so memory models which use this information should take `MemType_*Cached` as a signal to *not* register the upcall.

9.4.3 Byte-lane memory interface

For StrongARM (which externally can use a byte-lane memory interface), there is a variant of the basic memory interface. All the function types are the same, and the model should 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.

In place of `acc_WIDTH`, there is instead:

```
acc_BYTELANE(acc)
```

This returns a four-bit mask of which bytes in the word passed to the `access` function 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, and so on. `armflat.c` contains an example function that implements this model.

In `bytelane.c`, a model of an ASIC that converts the basic memory interfaces into the Byte-Lane version is provided as an example.

9.4.4 Other interfaces

There are other Memory Interface types defined, which are used internally by the ARM8 and StrongARM core models for communicating with their cache models.

9.4.5 Memory map handling

ARMulator does not directly support the `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 example 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.

9.5 Coprocessor Model Interface

The coprocessor model interface is defined in `armdefs.h`. Coprocessors are either initialized directly by the ARMulator as appropriate, or can be attached directly by another model by calling `ARMul_CoProAttach`:

```
ARMul_Error ARMul_CoProAttach(ARMul_State *state, unsigned number,
      const ARMul_CPInit *init, toolconf config, void *handle)
```

`init` is a pointer to a coprocessor initialization function

`handle` is a mechanism to allow the caller and the coprocessor to share state, and is passed into the coprocessor's initialization function

```
void *init(ARMul_State *state, unsigned num,
      ARMul_CPInterface *interf, toolconf config, void *handle)
```

As with memory models, the coprocessor `init` function must fill its handle and a table of access functions in `interf`. The `handle` argument is the value passed into `ARMul_CoProAttach`, if the coprocessor was initialized in that way. This function should either return zero or an error value.

```
unsigned ldc(void *handle, unsigned type, ARMword instr,
      ARMword value)
```

The `ldc` function is called whenever the ARMulator encounters an `LDC` instruction destined for this coprocessor. The `value` argument is the data loaded from memory. This function is called first with type set to `ARMul_FIRST`, is then called with type set to `ARMul_TRANSFER`, and finally with type set to `ARMul_DATA`, at which point `value` becomes valid. The function may request further data by returning `ARMul_INC`, in which case subsequent calls have type set to `ARMul_DATA` and `value` is valid.

```
unsigned stc(void *handle, unsigned type, ARMword instr,
      ARMword *value)
```

The `stc` function is called whenever the ARMulator encounters an `STC` instruction destined for this coprocessor. The `value` argument should be set to the value to be stored to memory. This location should be considered write-only.

This function is called first with type set to `ARMul_FIRST`, the function is then called with type set to `ARMul_TRANSFER`, and finally with type set to `ARMul_DATA`, at which point `value` becomes valid. The function may request further data by returning `ARMul_INC`, in which case subsequent calls have type set to `ARMul_DATA` and `value` is valid.

```
unsigned mrc(void *handle, unsigned type, ARMword instr,
      ARMword *value)
```

The `mrc` function is called whenever the ARMulator encounters an `MRC` instruction destined for this coprocessor. The `value` argument is a pointer to (the model of) an ARM register where the result of the transfer should be stored (if successful). This location should be considered as write-only. Unless it returns the value `ARMul_BUSY`, it will only be called once, with type set to `ARMul_FIRST`.

```
unsigned mcr(void *handle, unsigned type, ARMword instr,
      ARMword value)
```

The `mcr` function is called whenever the ARMulator encounters an `MCR` instruction destined for this coprocessor. The `value` argument is the value of the ARM register to transfer (if successful). Unless it returns the value `ARMul_BUSY`, it will only be called once with type set to `ARMul_FIRST`.

`unsigned cdp(void *handle, unsigned type, ARMword instr)`

The `cdp` function is called whenever the ARMulator encounters a `CDP` instruction destined for this coprocessor. Unless it returns the value `ARMul_BUSY`, it will only be called once with type set to `ARMul_FIRST`.

If a coprocessor does not handle one or more of these functions, it should leave their entries in the function table unchanged.

9.5.1 Function parameters

The argument `type` in the above functions can have one of five values:

`ARMul_FIRST`	indicates that this is the first time the coprocessor model has been called for this instruction
`ARMul_TRANSFER`	indicates that this is the load cycle of an `LDC` instruction (the data is being loaded from memory)
`ARMul_DATA`	indicates that the coprocessor is being called with valid data (`LDC`/`MCR`), or is expected to return valid data (`STC`/`MRC`)
`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_BUSY`	is used as the reply to a previous call that returned `ARMul_BUSY`

The `instr` parameter is the coprocessor instruction itself.

The functions must return one of four values:

`ARMul_BUSY`	indicates that the coprocessor is busy, and the ARMulator should busy-wait, calling the routine continuously
`ARMul_CANT`	indicates that the coprocessor cannot execute this particular instruction
`ARMul_INC`	indicates that the ARMulator should produce the next address for an `LDC`/`STC` instruction, and then call the model again
`ARMul_DONE`	indicates successful completion of the instruction

9.5.2 Debug functions

Two functions are provided that allow a debugger to read and write coprocessor registers via the Remote Debug Interface.

`unsigned read(void *handle, unsigned reg, ARMword *value)`

The `read` function reads the coprocessor register numbered `reg` and transfers its value to the location addressed by `value`.

`unsigned write(void *handle, unsigned reg, ARMword const *value)`

The `write` function sets the value of the coprocessor register numbered `reg` to the value addressed by `value`.

The function table also contains the entry `reg_bytes`, which describes to the ARMulator the configuration of the coprocessor's registers. It consists of an array of words:

- the first gives the number of registers
- the remaining vector gives the minimum number of words required to contain each register

For an example, see the definition of a minimal MMU system in the file `dummymmu.c`.

9.6 Operating System or Low-level Monitor Interface

Rapid prototyping of low-level operating system code is supported by ARMulator through an interface which allows a model to intercept SWIs and exceptions and model them on the host. As with other models, the operating-system model is called through an initialization function exported in a stub. The full interface is defined in `armdefs.h`.

```
ARMul_Error init(ARMul_State *state,
       ARMul_OSInterface *interf,
       toolconf config)
```

The initialization function is passed a vector of functions to fill in. It should also fill in its handle, and return an error code. The memory system is guaranteed to be operating at this time, and hence the operating system can read and write to the emulated memory using the routines in ***9.7.6 Memory Access Functions*** on page 9-21. It can also run initialization code (for example, the supplied Demon model will load and then initialize the floating-point emulation code), but for that initialization code to be able to use the operating system model, the model must have filled in the `handle` entry in the `ARMul_OSInterface` with a handle onto its state.

```
unsigned handle_swi(void *handle, ARMword number)
```

This function is passed in the SWI number (in `number`) of each `SWI` instruction executed by ARMulator, as it is executed, allowing support code to simulate operating system operations. This code can model as much of your operating system as you choose. This model can communicate with the emulated application by reading and writing the emulated ARM state using the routines described in ***9.7 Accessing the ARMulator's State*** on page 9-15.

The function may refuse to handle the SWI by returning `FALSE`, or the model may choose not to handle SWIs 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.

```
unsigned exception(void *handle, ARMword vector, ARMword pc)
```

This function is called whenever a hardware exception occurs. 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.

`vector` contains the address of the vector about to be executed:

0x00	Reset
0x04	Undefined Instruction
0x1C	Fast Interrupt (FIQ), and so on

`pc` contains the program counter (including the effect of pipelining) at the time the exception occurred

The function may choose to ignore the exception by returning `TRUE`, and ARMulator will continue from the instruction following the aborted instruction. A return value of `FALSE` causes the exception to occur normally.

9.6.1 Using the UnkRDIInfoUpcall

The `demon.c` model supplied with ARMulator uses the `UnkRDIInfoUpcall` in several places, 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

9.6.2 Using the floating-point emulator (FPE)

The ARMulator is supplied with the floating-point emulator (FPE) in object form. The Angel/Demon debug monitor model (`angel.c`) loads and starts executing the FPE on initialization. Note, however, that the supplied FPE requires the following SWIs to be supported by the debug monitor (Angel does not support these SWIs):

`SWI_Exit` (`0x11`)

`SWI_GenerateError`(`0x71`)

To load and initialize the FPE, call:

```
int ARMul_FPEInstall(ARMul_State *state)
```

This writes the FPE into memory (below 0x8000), and starts it running.

Note *Because this involves running code, it must only be done after the ARMulator is fully initialized. Before calling* `ARMul_FPEInstall`*, Demon completely initializes itself.*

```
int ARMul_FPEVersion(ARMul_State *state)
```

Returns the FPE version number. Demon uses this for unwinding aborts inside the emulator (see the `demon.c` source code for details).

```
int ARMul_FPEAddressInEmulator(ARMul_State *state, ARMword addr)
```

Returns TRUE if the stated address lies inside the FPE source.

9.7 Accessing the ARMulator's State

All the models have passed in a `state` variable which is an opaque handle onto the ARMulator's internal state. ARMulator exports a number of functions which allow models to access this state through this handle.

It is not sensible to access some parts of the state from certain parts of a model. For example, it is not sensible to set the contents of an ARM register from a memory access function, because this may be called midway through emulation of an instruction, whereas it would be sensible (and in fact necessary) to do so from a SWI handler function.

9.7.1 Accessing ARM registers

```
ARMword ARMul_GetReg(ARMul_State *state,
       unsigned mode,
       unsigned reg)
void ARMul_SetReg(ARMul_State *state,
       unsigned mode,
       unsigned reg,
       ARMword value)
```

These functions allow a register from a given mode to be read and written. Register 15 should not be accessed with these functions. Use `ARMul_GetPC`, `ARMul_SetPC`, `ARMul_GetR15`, and `ARMul_SetR15` described in this section.
The mode numbers are defined in `armdefs.h` as follows:

`USER26MODE`	`USER32MODE`
`FIQ26MODE`	`FIQ32MODE`
`IRQ26MODE`	`IRQ32MODE`
`SVC26MODE`	`SVC32MODE`
	`ABORT32MODE`
	`UNDEF32MODE`
	`SYSTEM32MODE`

In addition, the special value `CURRENTMODE` is defined. The function `ARMul_GetMode` returns the current mode number.

```
ARMword ARMul_GetPC(ARMul_State *state)
void ARMul_SetPC(ARMul_State *state,
       ARMword value)
ARMword ARMul_GetR15(ARMul_State *state)
void ARMul_SetR15(ARMul_State *state,
       ARMword value)
```

These functions allow access to Register 15. (If the processor is in a 26-bit mode, the PC variants strip the condition code and mode bits from register 15.)

```
ARMword ARMul_GetCPSR(ARMul_State *state)
void ARMul_SetCPSR(ARMul_State *state,
       ARMword value)
```

These functions allow the CPSR to be read or written (a valid value is faked if the processor is in a 26-bit mode, so these functions can still be used).

Note *For more information about 26-bit modes, see the ARM Architecture Reference Manual (ARM DDI 0100), 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.*

```
ARMword ARMul_GetSPSR(ARMul_State *state,
       ARMword mode)
void ARMul_SetSPSR(ARMul_State *state,
       ARMword mode,
       ARMword value)
```

Similarly, these functions allow a specific mode's SPSR to be read/written.

9.7.2 Accessing coprocessor registers

The state of coprocessor registers can be accessed using three functions.

```
unsigned int const *ARMul_CPRegWords(ARMul_State *state,
       unsigned cp)
```

This returns the `reg_words` array for the numbered coprocessor (see ***9.5 Coprocessor Model Interface*** on page 9-10 for details).

```
int ARMul_CPRead(ARMul_State *state,
       unsigned cp,
       unsigned reg,
       ARMword *value)
int ARMul_CPWrite(ARMul_State *state,
       unsigned cp,
       unsigned reg,
       ARMword const *value)
```

These functions call the `read` and `write` methods for a coprocessor. They also intercept calls to read/write the FPE's "emulated" registers. (See ***9.6.2 Using the floating-point emulator (FPE)*** on page 9-14.)

9.7.3 Interrupts

```
unsigned ARMul_SetNirq(ARMul_State *state,
       unsigned value)
unsigned ARMul_SetNfiq(ARMul_State *state,
       unsigned value)
unsigned ARMul_SetNreset(ARMul_State *state,
       unsigned value)
```

These three functions allow models to raise interrupts and cause a reset. They return the old settings of the signals. All three signals are reverse logic.

9.7.4 Configuration

```
ARMword ARMul_SetConfig(ARMul_State *state,
      ARMword changed,
      ARMword config)
```

This function allows a model to change the Configuration pins (see ***ConfigChangeUpcall*** on page 9-18 for a description of these pins and the bits assigned to each). Two bitfields are passed in:

- the first (`changed`) has bits set for each bit you wish to change
- the second (`config`) has the new values.

(A bit should not be clear in `changed` but set in `config`.) The `ConfigChange` upcalls will be called.

9.7.5 Upcalls

The ARMulator can be made to call back your model when some state values change. This can be used to avoid having to check these 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's **bigend** signal, so a memory model would attach to the `ConfigChange` upcall to be informed when this changes.

All the upcalls are defined in `armdefs.h`.

All the upcalls are called when the ARMulator resets and after ARMulator initialization is complete, regardless of whether the signals have changed, with the exception of `UnkRDIInfoUpcall`.

ExitUpcall

```
typedef void armul_ExitUpcall(void *handle)
```

All the exit upcalls are called when the ARMulator exits, and should be used to release any store used. Note that the ANSI `free` function is a valid `ExitUpcall`.

If no exit upcall is registered and a model uses some store, that memory will be lost.

```
typedef armul_InstallExitHandler(ARMul_State *state,
      armul_ExitUpcall *new,
      void *handle)
int ARMul_RemoveExitHandler(ARMul_State *state,
      void *node)
```

An upcall is installed using `armul_InstallExitHandler`. The `new` argument is the function to be called, and `handle` is the handle to be passed to it. It returns a handle to the callback, which can be passed to `ARMul_RemoveExitHandler` (as `node`) to remove the upcall.

ModeChangeUpcall

```
typedef void armul_ModeChangeUpcall(void *handle,
       ARMword old,
       ARMword new)
```

The mode change upcall is called whenever the ARMulator changes mode. The upcall is passed both the `old` and `new` modes. An enumeration of valid mode numbers is provided in `armdefs.h`.

An upcall is installed using:

```
void *ARMul_InstallModeChangeHandler(ARMul_State *state,
       armul_ModeChangeUpcall *new,
       void *handle)
```

in the same way as an ExitUpcall, and removed using:

```
int ARMul_RemoveModeChangeHandler(ARMul_State *state,
       void *node)
```

TransChangeUpcall

```
typedef void armul_TransChangeUpcall(void *handle,
       unsigned old,
       unsigned new)
```

The `nTRANS` change upcall is called when the **nTRANS** signal on the ARM core changes. **nTRANS** is the Not Memory Translate signal. When `LOW`, it indicates that the processor is in user mode, or that the processor is doing an `LDRT`/`STRT` instruction from a non-user mode. It may 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).

An upcall is installed using:

```
void *ARMul_InstallTransChangeHandler(ARMul_State *state,
       armul_TransChangeUpcall *new,
       void *handle)
```

and can be removed using:

```
int ARMul_RemoveTransChangeHandler(ARMul_State *state,
       void *node)
```

ConfigChangeUpcall

```
typedef void armul_ConfigChangeUpcall(void *handle,
       ARMword old,
       ARMword new)
```

The Config bits are those signals which are configuration pins on the ARM core. The words passed into the `ConfigChange` upcall are a bitfield of these signals, where each bit corresponds to a signal.

The bits are allocated as the bits in the System Coprocessor (coprocessor 15) Control Register:

`ARMul_Prog32`	(bit 4)	
`ARMul_Data32`	(bit 5)	
`ARMul_LateAbt`	(bit 6)	(not on ARM7, ARM8, StrongARM)
`ARMul_BigEnd`	(bit 7)	
`ARMul_BranchPredict`	(bit 11)	(ARM8 only)

An upcall is installed using:

```
void *ARMul_InstallConfigChangeHandler(ARMul_State *state,
       armul_ConfigChangeUpcall *new,
       void *handle)
```

and can be removed using:

```
int ARMul_RemoveConfigChangeHandler(ARMul_State *state,
       void *node)
```

InterruptUpcall

```
typedef unsigned int armul_InterruptUpcall(void *handle,
       unsigned int which)
```

This upcall is called whenever the ARM core *notices* an interrupt (not when it takes an interrupt) or reset. For example, this can be used by a memory model to reset its state, implement a wake-up, and so on. It is called at the start of the instruction or cycle (depending on the core being emulated) when the interrupt is noticed.

The `which` value is a bitfield encoding which interrupt(s) have been noticed.

bit 0	Fast interrupt (FIQ)
bit 1	Interrupt request (IRQ)
bit 2	Reset

The interrupt responsible may be removed using `ARMul_SetNirq` or `ARMul_SetNfiq`, in which case the ARM will not notice the interrupt. Reset cannot be removed.

An upcall is installed using:

```
void *ARMul_InstallInterruptHandler(ARMul_State *state,
       armul_InterruptUpcall *new,
       void *handle)
```

and can be removed using:

```
int ARMul_RemoveInterruptHandler(ARMul_State *state, void *node)
```

ExceptionUpcall

```
typedef unsigned int armul_ExceptionUpcall(void *handle,
      ARMword vector,ARMword pc, ARMword instr)
```

This upcall is called whenever the ARM processor takes an exception; for example, a data abort or a SWI. 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).

The arguments identify:

- the exception to be taken (as the address of the appropriate hardware vector)
- the PC value at the time the exception occurs
- the instruction that caused the exception

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" [9.3]).*

The sample models shipped with this release of ARMulator can be built either as a "Basic" model or as an "Operating-System" model.

An upcall is installed using:

```
void *ARMul_InstallExceptionHandler(ARMul_State *state,
      armul_ExceptionUpcall *new,
      void *handle)
```

and can be removed using:

```
int ARMul_RemoveExceptionHandler(ARMul_State *state,void *node)
```

UnkRDIInfoUpcall

```
typedef int armul_UnkRDIInfoUpcall(void *handle,
      unsigned type
      ARMword *arg1,
      ARMword *arg2)
```

The `UnkRDIInfoUpcall` can be used by a model extending the ARMulator's RDI interface with the debugger. An example of such a model is the profiler module (in `profiler.c`) which provides the `RDIProfile` info calls.

`UnkRDIInfoUpcall` functions are called if the ARMulator cannot handle an `RDIInfo` request itself. They return an `RDIError` value. The ARMulator will stop calling `UnkRDIInfoUpcall` functions once one returns a value other than `RDIError_UnimplementedMessage`.

Upcalls can be added using:

```
void *ARMul_InstallUnkRDIInfoHandler(ARMul_State *state,
   armul_UnkRDIInfoUpcall *proc,
   void *handle)
```

and can be removed using:

```
int ARMul_RemoveUnkRDIInfoHandler(ARMul_State *state,
   void *node)
```

In addition, the following UnkRDIInfo upcalls are called for the RDI Info calls:

`RDIInfo_Target`	This allows 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 (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.

Because these three calls have already been dealt with by the ARMulator, and are being passed around merely for information, or for all models to add information to the reply, models should always respond with `RDIError_UnimplementedMessage`, so that the message is passed on, even if they have responded in some way.

9.7.6 Memory Access Functions

The memory model may 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.

```
ARMword ARMul_ReadWord(ARMul_State *state,
       ARMword address)
ARMword ARMul_ReadHalfWord(ARMul_State *state,
       ARMword address)
ARMword ARMul_ReadByte(ARMul_State *state,
       ARMword address)
```

Returns the word/halfword/byte at the given address.

```
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)
```

Writes the specified word/halfword/byte at the specified address.

`unsigned long ARMul_ReadClock(ARMul_State *state)`

Returns a microsecond counter since start of emulation. (For example, this function might be used by an operating system model to provide a model of a system timer.) This calls the `read_clock` method of the installed memory model.

9.7.7 Event handling

The ARMulator has two types of events

- instructions
- cycles

Instructions

The ARMulator provides a mechanism for calling a function every instruction, or every *n* instructions (for a configurable value of *n*).

```
typedef void armul_Hourglass(void *handle,
       ARMword pc,
       ARMword instr)
```

The hourglass function is passed in the `pc` and instruction about to be executed. It is installed in the same way as upcalls, using:

```
void *ARMul_InstallHourglass(ARMul_State *state,
       armul_Hourglass *proc,
       void *handle)
```

There is a corresponding remove function:

```
int ARMul_RemoveHourglass(ARMul_State *state, void *node)
```

By default the function is called every instruction. However, you can change this by calling:

```
unsigned long ARMul_HourglassSetRate(ARMul_State *state,
       void *node,
       unsigned long rate)
```

which returns the old rate. The `rate` parameter passed in 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.

The `node` parameter is the handle returned from `InstallHourglass`.

Cycles

This event relies on the memory model accurately providing cycle counts.

The ARMulator has two routines to assist with scheduling cycle-based events:

`unsigned long ARMul_Time(ARMul_State *state)`

This function returns the number of core cycles executed since system reset.

```
typedef unsigned armul_EventProc(void *handle)
void ARMul_ScheduleEvent(ARMul_State *state,
      unsigned long delay,
      armul_EventProc *proc,
      void *handle)
```

This function allows a function (passed in the argument `proc`) to be called `delay` core cycles into the future, therefore allowing code such as multicycle FPU instructions to produce results sometime in the future. The function is called with the `handle` passed in. Note, however, that the function may only be called on the first instruction boundary following the specified cycle.

Because this works on core cycles, the results may not be as expected for cached processors. For example, core cycles may be a mix of cycles on a high-speed internal and a slower external clock. It is recommended that designs use their own cycle counting for scheduling events.

9.7.8 Miscellaneous functions

`void ARMul_SWIHandler(ARMul_State *state, ARMword address)`

This function should 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. When the `handle_swi` function returns (with the value TRUE to indicate the SWI has been handled) execution continues from the instruction at address in supervisor mode.

For an example of its use, see the code for handling `SWI_GenerateError` in `demon.c`.

`ARMword ARMul_SetMemSize(ARMul_State *state, ARMword size)`

This function should be called from memory initialization to specify where the top of memory is. (This value will eventually be used by the emulated ARM C runtime system to set up an application stack. Its value should not exceed `0x80000000`.)

`ARMword ARMul_GetMemSize(ARMul_State *state)`

This function is used by, for example, a debug monitor model to tell an application where the top of usable memory is, to set up application memory.

`ARMword ARMul_GetMode(ARMul_State *state)`

Return the current mode. If this is to be done frequently, a model should install a `ModeChange` upcall instead (see ***ModeChangeUpcall*** on page 9-18).

`ARMword ARMul_Properties(ARMul_State *state)`

Returns the properties word associated with the processor being emulated. This is a bitfield of properties, defined in `armdefs.h`.

`unsigned ARMul_CondCheckInstr(ARMul_State *state, ARMword instr)`

Given an instruction, this function returns `TRUE` if it would execute given the current state of the PSR flags.

9.7.9 Initialization errors

The model initialization functions return an `ARMul_Error` value. This can be one of the following:

`ARMulErr_NoError` (zero)	indicates a successful initialization
an `RDIError` value	from those in `dbg_rdi.h`
an `ARMulErr` value	from those in `errors.h`

The `errors.h` file can be extended by adding more errors. However, new errors *must* added at the end of the file.

Entries in this file are of the form:

```
ERROR(ARMulErr_OutOfMemory, "Out of memory.")
```

This declares an error message, `ARMulErr_OutOfMemory`, with the textual form `Out of memory`.

Errors returned from initialization functions should be passed via `ARMul_RaiseError`. For example:

```
interf->handle = (model_state *)malloc(sizeof(model_state));
if (interf->handle == NULL)
       return ARMul_RaiseHandle(state, ARMulErr_OutOfMemory);
```

`ARMul_RaiseError` is a printf-style variadic function, and the “textual form” can be a printf-style format string. For example, the `ARMulErr_MemTypeUnhandled` error message, used by memory models to reject an interface type that they do not understand, 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 returns an error message such as `Memory model 'Flat' incompatible with bus interface`.

The prototype of the `RaiseError` function is:

```
ARMul_Error ARMul_RaiseError(ARMul_State *state,
       ARMul_Error errcode, ...)
```

It returns the error code passed in, after formatting the error message.

9.7.10 Running code

`ARMword ARMul_DoProg(ARMul_State *state)`

This starts running the emulator at the current PC value.

`ARMword ARMul_DoInstr(ARMul_State *state)`

This single-steps the emulator for one instruction (steps into branches, exceptions, etc.).

```
void ARMul_HaltEmulation(ARMul_State *state,
      unsigned end_condition)
```

This function makes the emulator stop execution at the end of the current instruction, giving a reason code. The debugger interprets this end condition and gives a suitable message. The `end_condition` should be one of the `RDIError` error values defined in `dbg_rdi.h`, though not all of these errors are valid.

`unsigned ARMul_EndCondition(ARMul_State *state)`

This function returns the `end_condition` passed to `HaltEmulation`.

9.8 Accessing the Debugger

9.8.1 Input/output functions

Several functions are provided to display messages in the host debugger. Under armsd these print messages to the console; under the ARM Debugger for Windows these display messages to the relevant window.

`void ARMul_DebugPrint(ARMul_State *state, const char *format, ...)`

This function displays a message, in the RDI logging window under the ARM Debugger for Windows, or to the console under armsd.

`void ARMul_DebugPause(ARMul_State *state)`

This function waits for the user to press any key.

`void ARMul_ConsolePrint(ARMul_State *state, const char *format, ...)`
`void ARMul_PrettyPrint(ARMul_State *state, const char *format, ...)`

These functions display a message to the console window under the ARM Debugger for Windows, or to the console or output file (when using the `-o` option) under armsd. The `PrettyPrint` function formats the text and should be used for displaying startup messages.

`const Dbg_HostosInterface *ARMul_HostIf(ARMul_State *state)`

This function returns the channel back to the debugger, as defined in `dbg_hif.h`. An operating system model can make use of this to efficiently access the console window (under the ARM Debugger for Windows) or console (under armsd) without going through `ARMul_ConsolePrint`, and to receive user input.

`void *dbgarg`

This is an argument to be passed to the `dbg` functions.

`void dbgprint(void *arg, const char *format, va_list ap)`

This is a `vfprintf` equivalent (as used by `ARMul_DebugPrint`).

`void dbgpause(void *arg)`

Waits for the user to press any key.

`void *hostosarg`

This is an argument to be passed to the following functions.

`void writec(void *arg, int c)`

Writes a single character to the console window under the ARM Debugger for Windows, or to the console under armsd. (This is used by `ARMul_ConsolePrint`, and by the emulation of `SWI_WriteC` in `demon.c`, `SYS_WriteC` in `angel.c`.)

`int readc(void *arg)`

Reads a single character of input from the host debugger.

`int write(void *arg, char const *buffer, int len)`

Writes a stream of data to the console window under the ARM Debugger for Windows, or to the console under armsd.

```
char *gets(void *arg, char *buffer, int len)
```

Reads a string from the host debugger.

9.8.2 Miscellaneous functions

```
ARMword ARMul_RDILog(ARMul_State *state)
```

This function returns the value of the RDI logging level.

9.9 Events

The ARMulator has a mechanism for broadcasting and handling events. These events are merely an event number and a pair of words. The number is used to identify the event, and the semantics of the words depends on the precise event.

The core ARMulator generates some example events, defined in `armdefs.h`, which are listed below:

Event name	Word 1	Word 2
`CoreEvent_Reset`	-	-
`CoreEvent_UndefinedInstr`	PC value	instruction
`CoreEvent_SWI`	PC value	swi number
`CoreEvent_PrefetchAbort`	PC value	-
`CoreEvent_DataAbort`	PC value	aborting address
`CoreEvent_AddrExceptn`	PC value	aborting address
`CoreEvent_IRQ`	PC value	-
`CoreEvent_FIQ`	PC value	-
`CoreEvent_IRQSpotted`	PC value	-
`CoreEvent_FIQSpotted`	PC value	-
`CoreEvent_ModeChange`	PC value	new mode

Table 9-1: Events from ARM processor core

Event name	Word 1	Word 2
`MMUEvent_DLineFetch`	miss address	victim address
`MMUEvent_ILineFetch`	miss address	victim address
`MMUEvent_DTLBWalk`	miss address	victim address
`MMUEvent_ITLBWalk`	miss address	victim address
`MMUEvent_LineWB`	line address	-
`MMUEvent_DCacheStall`	address causing stall	address fetching

Table 9-2: Events from MMU and cache (not on StrongARM-110)

Event name	Word 1	Word 2
PUEvent_Full	next PC value	-
PUEvent_Mispredict	address of branch	-
PUEvent_Empty	next PC value	-

Table 9-3: Events from pre-fetch unit (ARM8-based processors only)

Additional modules can provide new event types, and they will be handled in the same way.

Events are caught by installing an `Event` upcall:

```
typedef void armul_EventUpcall(void *handle,
       unsigned int event,
       ARMword addr1,
       ARMword addr2)
void *ARMul_InstallEventUpcall(ARMul_State *state,
       armul_EventUpcall *trace,
       void *handle)
```

which, like other upcalls, can be removed:

```
int ARMul_RemoveEventUpcall(ARMul_State *state, void *node)
```

`ARMul_RaiseEvent` is used to invoke events, which are then passed to the user-supplied event upcalls:

```
void ARMul_RaiseEvent(ARMul_State *state,
       unsigned int event,
       ARMword word1,
       ARMword word2)
```

10 ARM Debugger

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.

10.1 Command Language

The following sections describe the commands available under the command-line based version of the debugger. For details of how to produce images with suitable debugging data, see ***Chapter 1, C Compilers*** and ***Chapter 2, Assembler***. Examples that demonstrate running programs under the command-line-based armsd are given in the *Software Development Toolkit User Guide (ARM DUI 0040)*.

`typewriter`	Shows command elements that you should type at the keyboard.
`typewriter`	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 etc.
`{}`	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. Note that many command names can be abbreviated; the braces here show what can be left out. There is one case where braces are required by the debugger; these are enclosed in quote marks in the syntax pattern.

10.1.1 Names used in syntax descriptions

These terms are used in the following sections for the command syntax descriptions.

`context`
: The program's activation state. See ***10.5.1 Variable names and context*** on page 10-25.

`expression`
: An arbitrary expression using the constants, variables and operators described in ***10.5.3 Expressions*** on page 10-26. It is either a low-level or a high-level expression, depending on the command.

  Low-level
  : 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
  : are arbitrary expressions using constants, variables and operators. Low-level symbols may be included in high-level expressions by preceding them with @.

  `list`, `find`, `examine`, `putfile`, and `getfile` require low-level expressions as arguments; all others require high-level expressions.

`location`
: A location within the program (see ***10.5.2 Program locations*** on page 10-26).

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 ***10.5.1 Variable names and context*** on page 10-25 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 10-1: Format descriptors*** shows some common 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), eg. 00018abc This can be used with any kind of pointer.
float	 %e %f %g	Only use this for floating-point results: Exponent notation, eg. 9.999999e+00 Fixed point notation, eg. 9.999999 General floating-point notation, eg. 1.1, 1.2e+06

Table 10-1: Format descriptors

10.2 Command-line Options

armsd is an interactive source-level debugger providing high-level debugging support for languages such as C, and low-level support for assembly language.

To invoke armsd, use the command:

```
armsd {options} image-name {arguments}
```

The options are listed below. Underlining is used to show the permitted abbreviations. The options must go before the image name. Anything after the image name is treated as a program argument.

`-help`	gives a summary of the armsd command-line options
`-little`	specifies that memory should be little-endian
`-big`	specifies that memory should be big-endian
`-processor` *name*	specifies the cpu type
`-nofpe` or `-fpe`	specifies whether the ARMulator should load the FPE on startup. When testing code compiled using the floating-point library, you may wish not 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 then quit when execution stops
`-i`*name*	adds *name* to the set of paths to be searched to find source files

10.2.1 Debuggee selection

This image supports: -REMote, -ARMUL, -RDP, -ADP

`-remote`	selects remote debugging; by default this will be ADP
`-adp`	selects remote debugging using ADP. Use this with the Angel Debug Monitor or EmbeddedICE version 2.0 onwards
`-armul`	select the software ARM Emulator (ARMulator)
`-rdp`	select remote debugging using RDP. Use this with the Demon Debug Monitor or EmbeddedICE, prior to version 2.0.

Using -rdp

If `-rdp` is chosen, the following device drivers are supported:

```
-SERIAL
-SERPAR
```

Port specification with RDP

`-port` *p* selects the serial port. *p* may be 1,2, or a device name

`-sport` *p* selects the serial port. *p* may be 1,2, or a device name

`-pport` *p* selects the parallel port. *p* may be 1,2, or a device name

Port specification with ADP

`-port` *expr* selects serial comms. *expr* can be any of:

```
1
2
device_name
s=1
s=2
s=device_name
```

To select serial and parallel comms, *expr* can be:

`s=`*n*`,p=`*m*

where *n* and *m* can be 1, 2 or a device name

To select ethernet comms, *expr* can be:

`e=`*id*

where *id* is the ethernet address of the target board

In the case of serial and/or parallel comms, the following" may be prefixed to the port expression. This switches off the heartbeat feature of ADP.

`h=0`

`-linespeed` *n* sets the line speed to *n* (ADP and RDP)

`-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 (see the *Software Development Toolkit User Guide (ARM DUI 0040)*). This is only valid with an `armsd.map` file.

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`.

10.3 Commands Overview

This section lists all armsd commands. They are first briefly listed in functional groups, and then are explained more fully in an alphabetical list.

The functional groups are:

- Accessing and executing programs
- Symbols
- Controlling execution
- Program context
- Low-level debugging
- Coprocessor support
- Profiling commands
- Miscellaneous

The semicolon character (;) separates two commands on a single line. Note that armsd 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.

10.3.1 Accessing and executing programs

Specifying the source directory

The variable `$sourcedir` is used to specify the directory which contains the program source files. It can be set using the command:

```
{let} $sourcedir = string
```

The string should be a valid directory name.

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.

10.3.2 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.

10.3.3 Controlling execution

`break`	adds breakpoints.
`call`	calls a procedure.
`istep`	steps through one or more instructions.
`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`	sets a watchpoint.

10.3.4 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.

10.3.5 Low-level debugging

`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.

10.3.6 Coprocessor support

The symbolic debugger's coprocessor support allows access to registers of a coprocessor through a debug monitor which 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.
`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.

10.3.7 Profiling commands

`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.

10.3.8 Miscellaneous 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.
`while`	is part of a multi-statement line.
`quit`	terminates the current symbolic debugger session and closes any open log or obey files.

10.4 Commands List

`!`

Any command whose first character is ! is passed to the host operating system for execution. This gives access to the command line of the host system without quitting the debugger.

`|`

Introduces a comment line.

`alias`

Defines, undefines or lists aliases. It allows you to define symbolic debugger commands:

```
alias {name {expansion}}
```

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:

```
alias n step
alias s step in
```

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.

`arguments`

Shows the arguments that were passed to the current, or other active procedure.

```
arguments {context}
```

If *`context`* is not specified, the current context is used (normally the procedure active when the program was suspended). Each argument’s name and current value is displayed.

`backtrace`

Prints information about all currently active procedures, starting with the most recent, or for a given number of levels, specified using `count`:

```
backtrace {count}
```

`break`

Specifies breakpoints at:

- procedure entry and exit
- lines
- statements within a line

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` breaks Thumb code

`/32` breaks ARM code

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 on locations, see ***10.5.2 Program locations*** on page 10-26.

`count` specifies the number of times the statement there 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.

`break` displays the program and source line at the breakpoint, unless a `do` clause is specified. 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`* .

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, referring to them by:*

`#`*`breakpoint_number`*

All breakpoints can also be deleted by referring to them by location.

`call`

Calls a procedure:

```
call{/size} location {(expression-list)}
```

where:

`size` specifies which code type to break:

- `/16` breaks Thumb code
- `/32` breaks ARM code

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.

`location` 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.
If the procedure (or function) returns a value, examine it using:

```
print $result for integer variables
print $fpresult for floating-point variables
```

`comment`

Writes a message to `stderr`:

```
comment message
```

`coproc`

Describes the register set of a coprocessor and specifies how the contents of the registers are formatted for display. The syntax is:

```
coproc cpnum {regdesc}*
```

`regdesc` may describe one register, or a range of registers that are accessed and are to be formatted uniformly:

```
rno{:rno1}  size access-specifiers access-values {displaydesc}*
```

where:

`size` is the register size (in bytes).

`access-specifiers` may comprise the letters:

- `R` the register is readable
- `W` the register is writable
- `D` the register is accessed through CPDT instructions (if this is not present, the register is accessed through CPRT instructions).

access-values the format of this option 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 the following table.

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:	
		w*n*	the whole of the nth word of the register value
		w*n*[*bit*]	bit *bit* of the nth word of the register value
		w*n*[*bit1*:*bit2*]	bits *bit1* to *bit2* inclusive of the nth word of the register value. *bit1* and *bit2* may be given in either order.
field '{' string {string} '}'*	*field*	must take one of the forms w*n*[*bit*] or w*n*[*bit1*:*bit2*] above. 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 w*n*[*bit*] or w*n*[*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.	

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
```

`context`

Sets the context in which the variable lookup occurs. It 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. If *`context`* is not specified, the context is reset to the active procedure:

```
context context
```

`cregisters`

Displays the contents of all readable registers of a coprocessor, in the format specified by an earlier `coproc` command:

```
cregisters cpnum
```

`cregdef`

Describes how the contents of a coprocessor are formatted for display.

```
cregdef cpnum rno displaydesc
```

`cwrite`

Writes to a coprocessor register. The syntax is:

```
cwrite cpnum rno val {val}*
```

Register *`rno`* of coprocessor *`cpnum`* must have been specified as writable; each *`val`* is an integer value and there must be one *`val`* item for each word of the coprocessor register.

`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. Low-level symbols are accepted by default:

```
examine {expression1} {, {+}expression2 }
```

The start address is given by *`expression1`*. 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 executed
- the address following the last address displayed by the last `examine` command, if the context has not changed since the last `examine` command was executed

The end address is specified in *`expression2`*, 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).

`find`

Finds all occurrences in memory of a given integer value or character string:

```
find expression1 {,expression2 {,expression3}} or
find string {,expression2 {,expression3}}
```

Low-level symbols are accepted by default.

expression2 and *expression3* specify the lower and upper bounds for the search. If *expression2* is absent, the base of the currently loaded image is used. If *expression3* is absent, the top (R/W limit) of the currently loaded image is used.

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`.

`fpregisters`

Displays the contents of the eight floating-point registers f0 to f7 and the floating-point processor status register (FPSR):

```
fpregisters [/format]
```

There are two formats for the display of floating-point registers, selected using the *format* switch.

The simpler form displays the registers and FPSR, and the full version includes detailed information on the floating-point numbers in the registers. The command:

```
fpregisters
```

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
```

The alternative command:

```
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
- D double
- E extended
- 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

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

go

Starts execution of the program. 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:

```
go {while expression}
```

If `while` is used, *`expression`* is evaluated when a breakpoint is reached. If *`expression`* evaluates to true (ie. non-zero), the breakpoint is not reported and execution continues.

getfile

Reads the contents of an area of memory from a file. The contents of the file are written to memory as a sequence of bytes, starting at the address which is the value of *`expression`*. Low-level symbols are accepted by default:

```
getfile filename expression
```

`help`

Displays a list of available commands, or help on a particular command. The help displayed 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 *`:

```
help {command}
```

`in`

Changes the current context by one activation level. 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.

`istep`

Steps execution through one or more instructions. 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:

```
istep {in} {count|w{hile} expression}
istep out
```

`language`

Sets the high-level language. 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`. In this case, `where` reports the current program counter and instruction; `step` steps by one instruction. If your program contains high-level debugging information and you wish to use low-level debugging, use:

```
language {none|C|F77|PASCAL|ASM}
```

`let`

Allows you to change the value of a variable or contents of a memory location:

```
{let} variable = expression{{,} expression}*
{let} memory-location = expression{{,} expression}*
```

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, since 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-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).

`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:

```
list{/size} {expression1}{, {+}expression2 }
```

where:

`size` distinguishes between ARM and Thumb code:

- `/16` lists Thumb code
- `/32` lists 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
- 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`*

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.

`load`

Loads an image for debugging:

```
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 program to be debugged.

`arguments` are the command-line arguments the program normally takes.

`image-file` and any necessary *`arguments`* may also be specified on the command-line when the debugger is invoked. 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.

`log`

Sends the output of subsequent commands to a file as well as to the screen:

```
log filename
```

where *`filename`* is the name of the file where the record of activity is being stored. 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.*

`lsym`

Displays low-level symbols and their values:

```
lsym pattern
```

where *`pattern`* is a symbol name or part of a symbol name. The wildcard (*) can be used at the beginning and/or end of the pattern to match any number of characters:

`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

`obey`

Executes a set of debugger commands which have previously been stored in a file, as if they were being typed at the keyboard:

```
obey command-file
```

where *`command-file`* is the name of the file containing the list of commands for execution.

`out`

Changes the current context by one activation level. The `out` command sets the context to be that of the caller of the current context.

Note *It is an error to issue an* `out` *command when no further movement in that direction is possible.*

pause

Prompts you to press a key to continue:

```
pause prompt-string
```

The prompt string is written to `stderr`, and execution continues only when a key is pressed. If you press ESC while commands are being read from a file, the file is closed before execution continues.

print

Examines the contents of the debugged program's variables, or displays the result of arbitrary calculations involving variables and constants:

```
p{rint}{/format} expression
```

The following example prints field `next` of structure `listp`:

```
print/%x listp->next
```

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.

profclear

Resets profiling counts:

```
profclear
```

profoff

Stops collecting profiling data:

```
profoff
```

profon

Starts collecting profiling data:

```
profon {interval}
```

The time between PC-sampling in microseconds is set by *`interval`*. Lower values have a higher performance overhead, and slow down execution, but higher values are not as accurate.

profwrite

Writes profiling information to a file:

```
profwrite {filename}
```

The generated information can be viewed using the `armprof` utility.

`putfile`

Writes the contents of an area of memory to a file which is written as a sequence of bytes:

`putfile` *`filename expression1, {+}expression2`*

The lower bound of the area of memory to be written is the value of *`expression1`*. The upper bound is the value of:

`expression2` `- 1` if *`expression2`* is not preceded by "+"

`expression1` `+` *`expression2`* `- 1` if *`expression2`* is preceded by "+"

Low-level symbols are accepted by default.

`quit`

Terminates the current symbolic debugger session and closes any open log or obey files:

`readsyms`

Loads debug information from a specified file (like the `symbols` option). The corresponding code must be present in another way (for example, via a getfile, or by being in ROM).

`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):

`registers {`*`mode`*`}`

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:

```
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
```

In addition to the mode names listed in ***10.7.1 Low-level symbols*** on page 10-33, *`mode`* may 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.

reload

Reloads the object file specified on the armsd command line, or the last load command:

```
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.

Breakpoints (but not watchpoints) remain set after a `reload` command.

return

Returns to the caller of the current procedure, passing back a result where required:

```
return {expression}
```

Note *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.*

step

Steps execution through one or more statements:

```
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.
- `count` specifies the number of statements to be stepped through: if it is omitted only one statement will be executed. The `while` clause continues single stepped execution until its *expression*, which is evaluated after every step, evaluates as false (zero).
- `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.

To step by instructions rather than statements, use the `istep` command, or enter:

```
language none
```

`symbols`

Lists all symbols defined in the given or current context, with their type information:

```
symbols {context}
```

The information produced is listed in the form:

```
name type, storage-class
```

To see global variables, use the filename with no path or extension as the context. To see internal variables, use `symbols $`

`type`

Types the contents of a source file, or any text file, between a specified pair of line numbers:

```
type {expression1} {, {{+}expression2} {,filename} }
```

The start line is given by *`expression1`*. 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
- the line following the last line displayed with the `type` command, if the context has not changed

The end line is given by *`expression2`*, 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`*

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.

`unbreak`

Removes a breakpoint:

```
unbreak {location}
```

`location` is either a source code location, or # followed by the breakpoint number, as displayed by `break`.

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.*

`unwatch`

Clears a watchpoint:

```
unwatch {variable}
```

`variable` can be either a variable name or the number of a watchpoint (preceded by #) set using `watch`. If only one watchpoint has been set, delete it using `unwatch`.

`variable`

Provides type and context information on the specified variable (or structure field).

```
variable variable
```

`variable` can also return the type of an expression.

`watch`

Sets a watchpoint on a variable. (Bitfields are not watchable.)

```
watch {variable}
```

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.

Notes *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.*

If EmbeddedICE is available, ensure that watchpoints use hardware watchpoint registers to avoid any performance penalty.

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 don't 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.

`where`

Prints the current context and shows the procedure name, line number in the file, filename and the line of code:

```
where {context}
```

If a context is specified after the `where` command, the debugger displays the location of that context.

`while`

This command is only useful at the end of an existing statement. You enter multi-statement lines by separating the statements with ";" characters:

```
statement; {statement;} while expression
```

Interpretation of the line continues until *`expression`* evaluates to false (zero).

10.5 Specifying Source-level Objects

10.5.1 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
```

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 (eg. 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 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
```

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
```

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
```

The complete syntax for the various ways of expressing context is:

```
{#}module{{:procedure}* {\{-}activation-number}}
{#}procedure{{:procedure}* {\{-}activation-number}}
#
```

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.

10.5.2 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 ***10.7.1 Low-level symbols*** on page 10-33 for further details.

Procedure entry and exit

Using a procedure name alone sets a breakpoint (see the `break` instruction on page 10-11) 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.

10.5.3 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.

Precedence	Operator	Purpose	Syntax
1	`()`	Grouping	`a * (b + c)`
	`[]`	Subscript	`isprime[n]`
	`.`	Record selection	`rec.field,a.b.c`
	`->`	Indirect selection (`rec->next` is identical to `(*rec).next`)	`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 nonprimitive values.*

10.5.4 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 “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.

10.6 Variables

This section lists the variables available in armsd, and gives information on manipulating them.

10.6.1 Summary of armsd variables

Many of the debugger's defaults can be modified by setting variables. Most of these are described elsewhere in this chapter in more detail:

`$clock`	number of microseconds since simulation started. This variable is read-only.
`$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).
`$format`	default format for printing integer values (initially set to "%ld").
`$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.
`$inputbase`	base for input of integer constants (initially set to 10).
`$list_lines`	default number of lines for list command (initially set to 16).
`$memory_statistics`	outputs any memory map statistics which the ARMulator has been keeping. This variable is read-only. See the *Software Development Toolkit User Guide (ARM DUI 0040)* for further details.
`$rdi_log`	rdi logging is enabled if non-zero, and serial line logging is enabled if bit 1 is set (initially set to 0).
`$result`	integer result returned by last "called" function (junk if none, or if an integer result was not returned). This variable is read-only.
`$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 only available when using EmbeddedICE version 2.00 onwards. It 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 (ie. 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:

A	address exception
D	data abort
E	Error
F	FIQ
I	IRQ
P	prefetch abort
R	reset
S	SWI
U	undefined instruction

10.6.2 armsd internal variables

The following variables are included to support EmbeddedICE.

`$icebreaker_lockedpoints`	shows or sets locked EmbeddedICE macrocell points
`$semihosting_enabled`	enables semihosting
`$semihosting_vector`	sets up semihosting SWI vector

Semihosting EmbeddedICE is described in the ***Software Development Toolkit User Guide (ARM DUI 0040)***.

Semihosting SWIs

There are also two variables for use when the debug agent is changing the semihosting SWIs it supports:

```
$semihosting_arm_swi
$semihosting_thumb_swi
```

These variables define how many ARM or Thumb SWIs are interpreted as semihosting requests by the debug agent.

In practice, these are used only in the ADP EmbeddedICE, as Angel only supports this as a recompilation option, and not at runtime. See ***Chapter 8, Angel*** for more information.

10.6.3 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, since 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).

10.6.4 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 section* ***10.1.1 Names used in syntax descriptions*** *on page 10-2.*

10.6.5 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.*

10.7 Low-level Debugging

Low-level debugging tables are generated automatically when programs are linked with the `-debug` flag set (this is enabled by default). In fact, it is not possible to include high-level debugging tables in an image without the low-level ones as well. There is no need to enable debugging at the compilation stage if only low-level debugging is to be done; just specify debugging when linking the program.

10.7.1 Low-level symbols

Low-level symbols are differentiated from high-level ones by preceding them with @.
A low-level symbol for a procedure refers to its call address, often the first instruction of the stack frame initialization, whereas the corresponding high-level symbol (if any) 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, with `watch` they stop execution if the word at the location named by the symbol changes.
Memory addresses can also be used with commands and should also be preceded by @.
Low-level symbols can also be used where a command would expect an expression; its value is the address of the low-level symbol.

Certain commands (`list`, `find`, `examine`, `putfile`, and `getfile`) accept low-level symbols by default. To specify a high-level symbol, precede it by "^".

Note *Low-level symbols do not have a context and so they are always available.*

10.7.2 Symbols for low-level entities

There are several predefined high-level naming low-level entities:

`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 (ie. this information is included in 26-bit address mode; not otherwise). 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, without any processor status register (PSR) flags.
`sp`	The stack pointer (r13).
`lr`	The link register (r14).
`fp`	The frame pointer (r11).
`psr` `cpsr`	`psr` and `cpsr` are synonyms for the current mode's processor status register.

`spsr` `spsr` is the saved status register for the current mode. 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 and ABORT32. 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*.

`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).

`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).

All these registers can be examined with the `print` command and changed with the `let` command. For example, the form `print/%x psr` displays the processor status register (PSR).

These symbols are defined in the root context, so if you have a variable r0 and you wish to refer to register 0, you can use # to specify the register as follows:

```
print #r0
```

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 26-bit supervisor mode, by typing:

```
let psr = %NvF_SVC26
```

Note *The percentage sign must precede the condition flags and the underscore which in turn must precede the processor mode description.*

10.8 armsd commands for EmbeddedICE

`listconfig` *`file`*
Lists the configurations known to the debug agent.

`loadagent`
Downloads a replacement EmbeddedICE ROM image, and starts it (in RAM):

`loadconfig` *`file`*
Loads an EmbeddedICE configuration data file:

`readsyms` *`file`*
Loads an image file containing debug information but does not download the image.

The highest-numbered version meeting the `version` constraint is used. For more information, see the *Software Development Toolkit User Guide*.

`selectconfig` *`name version`*
Selects an EmbeddedICE configuration to use, 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 (default)
- `n` uses version n
- `n+` uses version n or later

Debug communications channel

armsd accesses the debug communication channel using the following commands:

`ccin` *`filename`*
Selects a file containing Comms Channel data for reading.
This command also enables Host to Target Comms Channel communication.

`ccout` *`filename`*
Selects a file where Comms Channel data is written.
This command also enables Target to Host Comms Channel communication.

For more information, see *Application Note 38: Using the ARM7TDMI's Debug Communication Channel*.

10.9 Angel and armsd

For information on Angel, see ***Chapter 8, Angel***. The Angel-aware version of armsd supports shared device communications between the host and target application, accessed through a prefix command, `sys`.

This provides access to extended commands which support operating systems or special hardware features. The syntax is:

```
sys command args
```

The following commands are available:

`sys`
: Displays a list of installed commands. This is the same as typing:

```
sys help
```

`sys help`
: Displays a list of available SYS commands.

`sys help` *`command`*
: Displays help on a particular command.

`sys help *`
: Displays information about all SYS commands in addition to their names.

10.9.1 Angel SYS commands

The default Angel-aware version of armsd has the following `sys` commands shown below.

These commands control communications between the host and the target application via the shared device mechanism. This allows host and target to communicate using the same link that is being used by armsd/Angel.

The target application accesses the shared device via the API provided by `devappl.h` in the Angel sources.

`sys applin`
: Controls comms in the host to target application direction. The syntax is:

```
sys applin filename
```

With a filename, host to target comms is enabled and input for delivery to the target is taken from the named file or pipe. Without a filename, host to target comms is disabled.

`sys applout`
: Controls communications from the target application to the host via the shared device mechanism. The syntax is:

```
sys applout filename
```

With a filename, target to host comms is enabled and output from the target is written to the named file or pipe. Without a filename, target to host comms is disabled.

11 Remote Debugging

This chapter describes the Remote Debug Interface and the Angel Debug Protocol.

11.1 ARM Remote Debug Interface

This chapter describes a C interface to the Remote Debug Interface (RDI). The RDI is a procedural interface between a debugger and a debuggee via a debug monitor or controlling debug agent. The interface is designed to make it easier to use the RDI from a C program when the debugger and debug agent are linked as one program.

The interface can be pulled apart to yield a pair of *stub* interfaces communicating via the Angel Debug Protocol (for details see ***11.4 Angel Debug Protocol (ADP)*** on page 11-21).

The RDI gives the ARM symbolic debugger core a uniform way to communicate with:

- a controlling debug agent or debug monitor linked with the debugger
- a debug agent executing in a separate operating system process
- a debug monitor running on ARM-based hardware accessed via a communication link
- a debug agent controlling an ARM processor via hardware debug support

The RDI is not an entity fixed for all time. As it evolves, new levels of specification are added, and within any level of specification there are implementation options. This approach is taken so that a variety of minimal debug monitors and controlling debug agents can be accommodated without excessive overhead, and to ensure compatibility between debuggers and debug monitors released at different times. As a result, a debugger using the RDI must *negotiate* to establish its debuggee's capabilities and must not use capabilities its debuggee does not support.

Every function returns an error status. Zero indicates no error, otherwise the value returned is the error number (see ***11.3 Error Codes*** on page 11-19). It is the caller's responsibility to ensure that memory pointers point to valid memory locations in the debugger's address space.

Structure 1 arises in the variant of `armsd` which is linked with ARM's standard ARM emulation environment (for the PC- and Sun-hosted cross-development variants of `armsd`), and in the self-hosted, single address-space variant of `armsd` (for Acorn's RISC OS).

Structure 2 arises in an ARM-UNIX-hosted variant of `armsd`, if *armsd* and the ARM emulator (the ARMulator) were run in separate UNIX processes (perhaps on separate machines). In this case, the *RDI* would consist of two stubs using UNIX's remote procedure calls to effect the inter-process message passing.

Structures 3 and 4 arise when `armsd` is used to control a debuggee, executing on ARM-based hardware (for instance on the *Platform Independent Evaluation (PIE)* card) connected to `armsd`'s host via a hardware debugging channel; for example, via RS232 as used on the *PIE* card.

11.2 RDI Functions

The following list shows all the RDI functions available. Each function is described in detail on the following pages.

Function name	Purpose
`RDI_open`	open and/or initialize debuggee
`RDI_close`	close and finalize debuggee
`RDI_read`	read memory address
`RDI_write`	write memory address
`RDI_CPUread`	read cpu state
`RDI_CPUwrite`	write cpu state
`RDI_CPread`	read co-processor state
`RDI_CPwrite`	write co-processor state
`RDI_setbreak`	set breakpoint
`RDI_clearbreak`	clear breakpoint
`RDI_setwatch`	set watchpoint
`RDI_clearwatch`	clear watchpoint
`RDI_execute`	execute
`RDI_step`	multiple step
`RDI_pointinquiry`	break/watch inquiry
`RDI_addconfig`	add config block
`RDI_loadconfig`	load config block
`RDI_selectconfig`	select config block
`RDI_drivernames`	get driver names
`RDI_cpunames`	get cpu names
`RDI_errmess`	get error messages
`RDI_loadagent`	load debug agent
`RDI_info`	miscellaneous info

RDI_open (open and/or initialize debuggee)

```
int RDI_open(unsigned type, struct Dbg_ConfigBlock
             const *config, struct Dbg_HostasInterface
             const *i, struct Dbg_MCState *dbg_state)
```

This function opens or initializes the debugger, where:

`type` distinguishes between types of initialization:

- Bit 0 = 0 cold start (execute bootstrap, initialize MMU etc.)
- Bit 0 = 1 warm start (terminate execution, reset processor state, and so on)
- Bit 1 = 1 reset the communication link

`config` holds information such as the memory size, byte sex, serial port, processor, etc. See `dbg_conf.h` for full details.

`i` provides various functions which can be called to interact with the host's operating system, eg. print character to screen, read character from keyboard. See `dbg_hif.h` for full details.

`dbg_state` is internal to the debugger toolbox.

RDI_close (close and finalize debuggee)

```
int RDI_close()
```

This function terminates the current debugging session. Only a call to `RDI_open` may follow this call.

RDI_read (read memory address)

```
int RDI_read(unsigned long source, void *dest,
             unsigned *nbytes)
```

This function transfers data from the debuggee's memory to the debugger. Bytes are read from the debuggee at address `source`, and stored at location `dest` in the caller's address space. `nbytes` points to the number of bytes to transfer.

On return, the location pointed to by `nbytes` contains the number of bytes that were successfully transferred. If an error occurs, the state of the memory at `dest` is undefined.

RDI_write (write memory address)

```
int RDI_write(void *source, unsigned long dest,
              unsigned *nbytes)
```

This function transfers data from address `source` in the debugger to address `dest` in the debuggee. `nbytes` points to the number of bytes to transfer. On return, the location pointed to by `nbytes` contains the number of bytes that were successfully transferred. If an error occurs, the state of the memory at `dest` is undefined.

RDI_CPUread (read CPU state)

```
int RDI_CPUread(unsigned mode, unsigned long mask,
                unsigned long state[])
```

This function allows the debugger to read the values of the debuggee's CPU registers, where:

`mode` defines the ARM processor mode from which the transfer should be made. A value of `RDIMode_Curr` indicates that the prevailing processor mode should be used. Other values correspond to the mode the target ARM would be in if the mode bits of the PSR were set to this value.

`mask` indicates which registers should be transferred. Bit 0 of this word corresponds to register 0, bit 14 corresponds to the link register, and bit 15 the Program Counter (including the mode and flag bits in 26-bit modes). Other values can be ORed into the mask to retrieve other registers:

- `RDIReg_PC` to get just the Program Counter value
- `RDIReg_CPSR` to get the value of the CPSR
- `RDIReg_SPSR` to get the value of the SPSR in non-user modes

Notice that the value of Program Counter that is returned (via either bit 15 or `RDIReg_PC`) has already had the effect of pipelining removed, so it is eight less than the actual value in the Program Counter.

`state` is a pointer to enough words of memory in which to save the CPU state (four bytes per register). The requested registers are saved contiguously into this memory.

RDI_CPUwrite (write CPU state)

```
int RDI_CPUwrite(unsigned mode, unsigned long mask,
                 unsigned long state[])
```

This function allows the debugger to set the values of the debuggee's CPU registers. The arguments are as for `RDI_CPUread`, except that register values are read from `state` and written to the debuggee's register set.

RDI_CPread (read co-processor state)

```
int RDI_CPread(unsigned CPnum, unsigned long mask,
               unsigned long state[])
```

This function allows the debugger to read the debuggee's co-processor registers (it has a similar function to `RDI_CPUread`, except that the register values are taken from the co-processor whose number is specified by the `CPnum` argument). The actual registers transferred, and their size, depend on the co-processor specified. The transferred values are written to `state`.

By convention, the following co-processors are understood:

- Co-processor 1 (and 2 in the case of FPA) is a floating-point unit:

  Bits 0 to 7 of `mask` request the transfer of data from co-processor registers 0 to 7.
  Bit 8 designates the floating-point status register (FPSR).
  Bit 9 designates the floating-point command register (FPCR).
- Co-processor 15 is a memory management unit (eg. ARM3 or ARM600):

  Bits 0 to 7 of `mask` request transfer of MMU registers 0 to 7.

RDI_CPwrite (write co-processor state)

```
int RDI_CPwrite(unsigned CPnum, unsigned long mask,
                unsigned long state[])
```

This function allows the debugger to write the values of the debuggee's co-processor registers (it has a similar function to `RDI_CPUwrite`, except that the register values are written to the co-processor whose number is given by `CPnum`). The actual registers transferred, and their size, depend on the co-processor specified. The transferred values are read from `state`. Currently the following co-processors are understood:

- Co-processor 1 (and 2 in the case of FPA) is a floating-point unit:

  Bits 0 to 7 of `mask` request transfer of data to co-processor registers 0 to 7.
  Bit 8 designates the floating-point status register (FPSR).
  Bit 9 designates the floating-point command register (FPCR).
- Co-processor 15 is a memory management unit (eg. ARM3 or ARM600).

  Bits 0 to 7 of `mask` request transfer to MMU registers 0 to 7.

RDI_setbreak (set breakpoint)

```
int RDI_setbreak(unsigned long address, unsigned type,
                 unsigned long bound, PointHandle *point)
```

This function requests the debuggee to set an execution breakpoint at `address`.
If a breakpoint is set on a location which already has a breakpoint, the first breakpoint will be removed before the new breakpoint is set.

The `type` argument defines the sort of breakpoint to set:

`RDIPoint_EQ`	halt execution if the pc is equal to `address`.
`RDIPoint_GT`	halt execution if the pc is greater than `address`.
`RDIPoint_GE`	halt execution if the pc is greater than or equal to `address`.
`RDIPoint_LT`	halt execution if the pc is less than `address`.
`RDIPoint_LE`	halt execution if the pc is less than or equal to `address`.
`RDIPoint_IN`	halt execution if the pc is in the address range from `address` to `bound`, inclusive.
`RDIPoint_OUT`	halt execution if the pc is not in the address range `address` to `bound`, inclusive.
`RDIPoint_MASK`	halt execution if `(pc & bound) = address`.

Note the following bit settings of `type`:

Bit 4	If set, this indicates that the breakpoint is on a 16-bit (Thumb) instruction rather than a 32-bit (ARM) instruction.
Bit 5	indicates whether the breakpoint should be *conditional*. If it is set, execution halts only when the breakpointed instruction is executed, not when the condition code causes it to be skipped. Otherwise, breakpoints are unconditional: execution halts when the breakpoint is reached, regardless of the condition field of the breakpointed instruction.
Bits 6 and 7	are not used in the RDI, although they are used in the RDP. This is because the RDI supports these facilities directly.

If the call succeeds, `point` is set to a value which identifies the breakpoint. At RDI specification level 0, a breakpoint is identified by its address (the value of `address`); at levels 1 and above, it is identified by a handle returned by the debuggee (see the section ***RDI_info (miscellaneous information)*** on page 11-11).

Return values: A special return value, `RDIError_NoMorePoints`, indicates that the call to `RDI_setbreak` was successful but that there are no more breakpoint resources of this type available.

The return value `RDIError_CantSetPoint` shows that the call failed because the debuggee currently has insufficient breakpoint resources available to honour this request.

RDI_clearbreak (clear breakpoint)

```
int RDI_clearbreak(PointHandle point)
```

This function clears the execution breakpoint identified by `point` which was set by a previous call to `RDI_setbreak`.

RDI_setwatch (set watchpoint)

```
int RDI_setwatch(unsigned long address, unsigned type,
                 unsigned datatype, unsigned long bound,
                 PointHandle *point)
```

This function gets a data access watchpoint at `address` in the debuggee. If a watchpoint is set on a location which already has a watchpoint, the first watchpoint is removed before the new watchpoint is set.

Values may be summed or ORed together in order to halt on any of a set of memory access types. For example, to watch for any write access to the specified location(s):

```
RDIWatch_ByteWrite
RDIWatch_HalfWrite
RDIWatch_WordWrite
```

If the call succeeds, `*point` is set to a value which identifies the watchpoint to the debugee. At RDI specification level 0, a watchpoint is identified by its address (the value of `address`); at levels 1 and above it is identified by a handle returned by the debuggee (see the section ***RDI_info (miscellaneous information)*** on page 11-11).

`type` defines the type of watchpoint to set:

`RDIPoint_EQ`	halts on a data access equal to `address`
`RDIPoint_GT`	halts on a data access greater than `address`
`RDIPoint_GE`	halts on a data access greater than or equal to `address`
`RDIPoint_LT`	halts on a data access less than `address`
`RDIPoint_LE`	halts on a data access less than or equal to `address`
`RDIPoint_IN`	halts on a data access in the range from `address` to `bound`, inclusive
`RDIPoint_OUT`	halts on a data access not in the range from `address` to `bound`, inclusive
`RDIPoint_MASK`	halts execution if `(data-access-addr & bound) = addr`

`datatype` defines the type of data access to watch for:

`RDIWatch_ByteRead`	watches for byte reads
`RDIWatch_HalfRead`	watches for halfword reads
`RDIWatch_WordRead`	watches for word reads
`RDIWatch_ByteWrite`	watches for byte writes
`RDIWatch_HalfWrite`	watches for halfword writes
`RDIWatch_WordWrite`	watches for word writes

Return values: A special return value, `RDIError_NoMorePoints`, indicates that the call to `RDI_setwatch` was successful, but that there are no more watchpoint resources of this type available. The return value `RDIError_CantSetPoint` indicates that the call failed because the debuggee currently has insufficient watchpoint resources to honour this request.

RDI_clearwatch (clear watchpoint)

```
int RDI_clearwatch(PointHandle point)
```

This function clears the data access watchpoint identified by `point` which was set by a previous call to `RDI_setwatch`.

RDI_execute (execute)

```
int RDI_execute(PointHandle *point)
```

This function initiates execution in the debuggee, at the address currently loaded into the CPU Program Counter. This function returns an error code if:

- a breakpoint is reached
- a watched address is accessed
- an exception occurs
- you press Escape

See ***11.3 Error Codes*** on page 11-19 for further information.

If a breakpoint or watchpoint caused the return, `*point` is set to the handle identifying the break/watchpoint. At RDI specification level 0, a breakpoint/watchpoint is identified by its address; at levels 1 and above it is identified by a handle returned by the debuggee when the point was set.

RDI_step (multiple step)

```
int RDI_step(unsigned ninstr, PointHandle *point)
```

This function initiates execution in the debuggee at the address currently loaded into the CPU Program Counter, but only executes the number of instructions specified by `ninstr`.

If `ninstr` is zero, the debuggee executes instructions up to the next instruction that explicitly alters the program counter (ie. a branch or ALU operation with the program counter as destination).

If a breakpoint is reached, or a watched address is accessed, or an exception occurs, or you press Escape, or the end of the program is reached before `instr` instructions have been executed, `RDI_step` returns an error code indicating why execution was suspended (see ***11.3 Error Codes*** on page 11-19).

If a breakpoint or watchpoint caused the return, `*point` will be set to the handle identifying the breakpoint/watchpoint. At RDI specification level 0, a breakpoint/watchpoint is identified by its address; at levels 1 and above it is identified by a handle returned by the debuggee when the breakpoint/watchpoint was set.

RDI_pointinquiry (breakpoint/watchpoint inquiry)

```
int RDI_pointinquiry(unsigned long *address, unsigned type,
                     unsigned datatype,unsigned long *bound)
```

This function returns information about what happens if a corresponding call is made to `setbreak` or `setwatch`. (For range and comparison point types, the debuggee tries to honour the request and is not required to use precisely the address and bound requested.)

If the break/watch type is supported, `address` and, if applicable, `bound` are updated to the values that will be used if a breakpoint or watchpoint is set.

If the corresponding breakpoint/watchpoint request cannot be honored because there are no breakpoint/watchpoint resources left, the value `RDIError_NoMorePoints` is returned.

For inquiries about breakpoints, `datatype` must be 0. Otherwise, `type` and `datatype` are precisely as in corresponding calls to `setbreak` and `setwatch`.

Note *The absence of a return value of* `RDIError_NoMorePoints` *from* `setbreak` *or* `setwatch` *does not mean that the next request can be honoured, but merely that there is some value of* `type` *and* `datatype` *for which a following request can be honoured. To be sure that a request will be honored, it is necessary to call* `RDI_pointinquiry`*.*

RDI_addconfig (add config block)

```
int RDI_addconfig(unsigned long bytes)
```

This function declares the size of a config block about to be loaded.

RDI_loadconfig (load config block)

```
int RDI_loadconfig(unsigned long nbytes, char const *data)
```

This function loads the config block of size `nbytes`, pointed to by `data`. This config block specifies target-dependent information to the Debug Agent. See the documentation on the Debug Agent concerned for more detail (for example, ICEman).

RDI_selectconfig (select config block)

```
int RDI_selectconfig(RDI_ConfigAspect aspect, char *const name,
                     RDI_ConfigMatchType matchtype,
                     unsigned versionreq, unsigned *versionp)
```

This function selects which of the loaded config blocks should be used, and then re-initializes the Debug Agent to use the selected configuration data.

`aspect`	is one of `RDI_ConfigCPU` or `RDI_ConfigSystem`
`name`	is the name of the configuration to be selected

`matchtype`	specifies how the version number must match that requested:
	`RDI_MatchAny`
	`RDI_MatchExactly`
	`RDI_MatchNoEarlier`

`versionreq`	is the version number requested
`versionp`	is the version actually selected

RDI_NameList (get driver names)

```
RDI_NameList const *RDI_drivernames(void)
```

This function is `typedef`'d to be a `struct` containing the number of names and an array of these names. The returned names are used to recognize whether a particular driver has been selected on the command line.

RDI_NameList (get CPU names)

```
RDI_NameList const *RDI_cpunames(void)
```

This function works in a similar way to `RDI_DriverNames`.

RDI_ErrMess (get error messages)

```
int RDI_ErrMess(char *buf,int buflen,int errno)
```

This function requests that an error message (up to `buflen` characters) corresponding to `errno` is placed in buffer `buf`.

RDI_loadagent (load debug agent)

```
int RDI_loadagent(ARMword dest, unsigned long size,
                  getbuffer proc *getb, void *getbarg)
```

This function downloads a new version of the Debug Agent. It can be used only if `RDIInfo_Target` returns a value with `RDITarget_LoadAgent` set.

`dest` address in the Debug Agent's memory where the new version will be put

`size` size of the new version, in bytes

`getb` a function which can be called (with `getbarg` as the first argument) and the number of bytes to download this call as the second argument

11.2.1 Miscellaneous functions

RDI_info (miscellaneous information)

```
int RDI_info(unsigned type, unsigned long *arg1, unsigned long *arg2)
```

This function is used to transfer miscellaneous information between the debugger and the debuggee. Not all types make use of all three arguments. The information transferred is dependent on the value of the first argument. A status value is returned to indicate success or failure of the call.

RDIInfo_Points (breakpoints and watchpoints)

```
int RDI_info(RDIInfo_Points, unsigned long *arg1)
```

After a call of type `RDIInfo_Points`, the word addressed by `arg1` should be interpreted as a set of bits as follows:

Bit 0: comparison breakpoints/watchpoints are supported

Bit 1: range breakpoints/watchpoints are supported

Bit 2: watchpoints for byte reads are supported

Bit 3: watchpoints for halfword reads are supported

Bit 4: watchpoints for word reads are supported

Bit 5: watchpoints for byte writes are supported

Bit 6: watchpoints for halfword writes are supported

Bit 7: watchpoints for word writes are supported

Bit 8: mask breakpoints/watchpoints are supported

Bit 9: thread-specific breakpoints are supported

Bit 10: thread-specific watchpoints are supported

Bit 11: conditional breakpoints are supported

Bit 12: status enquiries about the capabilities of (H/W) breakpoints and watchpoints are allowed

If none of bits 2–7 are set, bits 0, 1, and 8 apply only to breakpoints. Otherwise, bits 0, 1, and 8 apply to both breakpoints and watchpoints

Note *All debuggees support breakpoints of type* `RDIPoint_EQ` *(break at specified address), so there is no bit denoting this in the value returned by* `RDIInfo_Points`.

RDIInfo_Step

```
int RDI_info(RDIInfo_Step, unsigned long *arg1)
```

After a call of type `RDIInfo_Step`, the location addressed by `arg1` should be interpreted as follows:

Bit 0: single stepping of more than one instruction is supported

Bit 1: single stepping to the next direct PC alteration is supported

Bit 2: single stepping of a single instruction is supported

RDIInfo_Target (identify target)

```
int RDI_info(RDIInfo_Target, unsigned long *arg1, unsigned long *arg2)
```

After calling `RDIInfo_Target`, the value addressed by `arg1` should be interpreted as:

Bit 16: 1 => the debuggee has a communications channel

Bit 15: 1 => can cope with 16-bit (Thumb) code

Bit 14: 1 => the Debug Agent can do profiling

Bit 13: 1 => understands `RDPInterrupt` (a single-byte version of `RDI_SignalStop`)

Bit 12: 1 => asks about the maximum size download chunk the agent can accept

Bit 11: 1 => the Debug Agent can be reloaded

Bits 8,9,10: the minimum RDI specification level (0–7) that the debugger requires of the debuggee

Bits 5,6,7: the maximum RDI specification level (0–7) implemented by the debuggee

Bit 4 = 0: debuggee is running under a software emulator

Bit 4 = 1: debuggee is running on ARM hardware

Bits 0:3: host speed as 10**(bits 0:3) instruction per second (IPS)
(0 => 1IPS, 1 => 10IPS, 2 => 100IPS, 3 = 1000IPS, ..., 6 => 1MIPS, ...)

The value addressed by `arg2` is a unique identifier, which identifies the ARM processor or ARM emulator under which the debuggee is running.

Bits 5..10 allow a debugger to negotiate a suitable RDI specification level with a debuggee or to report that it is incompatible with the debuggee.

RDIVector_Catch

```
int RDI_info(RDIVector_Catch, unsigned long *arg1)
```

A set bit in the location addressed by `arg1` indicates to the debuggee that the corresponding exception should be reported to the debugger, as follows:

Bit 0:	Branch through 0	Bit 5:	Address exception
Bit 1:	Undefined instruction	Bit 6:	Interrupt (IRQ)
Bit 2:	Software interrupt (SWI)	Bit 7:	Fast interrupt (FIQ)
Bit 3:	Prefetch abort	Bit 8:	Error
Bit 4:	Data abort		

Bits which are 0 indicate that the corresponding exception vector should be taken by the debuggee.

RDIInfo_MMU

```
int RDI_info(RDIInfo_MMU, unsigned long *arg1, unsigned long *arg2)
```

This enquires about type and status of any MMU present. On return, `arg1` contains a word which identifies the types of the MMU. `arg2` addresses a word describing the status of the MMU.

RDIInfo_DownLoad

```
int RDI_info(RDIInfo_DownLoad, unsigned long *arg1, unsigned long *arg2)
```

This enquires whether configuration download and selection is available. The status returned is OK if these features are available.

RDIInfo_SemiHosting

```
int RDI_info(RDIInfo_SemiHosting, unsigned long *arg1,
             unsigned long *arg2)
```

This enquires whether `RDISemiHosting_* RDIInfo` calls are available. The status returned is OK if these features are available.

RDIInfo_CoPro

```
int RDI_info(RDIInfo_CoPro, unsigned long *arg1, unsigned long *arg2)
```

This enquires whether the `CoPro RDI Info` calls are available. The status returned is OK if these features are available.

RDIInfo_Icebreaker

```
int RDI_info(RDIInfo_Icebreaker, unsigned long *arg1,
             unsigned long *arg2)
```

This enquires whether the debuggee is controlled by EmbeddedICE. The status returned is OK if the debuggee is controlled by EmbeddedICE.

RDIMemory_Access

```
int RDI_info(RDIMemory_Access, unsigned long *arg1, unsigned long *arg2)
```

This asks for the memory access statistics for a block of memory indicated by the handle addressed by `arg1`. On return `arg1` points to the memory access statistics. For more details, see `RDI_MemAccessStats` in `dbg_stat.h`.

RDIMemoryMap

```
int RDI_info(RDIMemoryMap, unsigned long *arg1, unsigned long *arg2)
```

This call sets the characteristics for an area of memory. For full details of memory descriptions, see `RDI_MemDesc` in `dbg_stat.h`. On entry:

`arg1` points to an array of memory descriptions (*n*)

`arg2` points to a word holding *n*.

RDISet_CPUSpeed

```
int RDI_info(RDISet_CPUSpeed, unsigned long *arg1)
```

This call sets the simulated CPU speed to be `arg1` (in nanoseconds).

RDIRead_Clock

```
int RDI_info(RDIRead_Clock, unsigned long *arg1, unsigned long *arg2)
```

This call reads the simulated CPU time. On return:

`arg1` addresses the time in nanoseconds

`arg2` addresses the time in seconds

RDIConfig_Count

```
int RDI_info(RDIConfig_Count, unsigned long *arg1)
```

This requests that `arg1` returns the number of configuration blocks known to the debug agent to the word address. Use this option only if `RDIInfo_Download` returned no errors.

RDIConfig_Ntl

```
int RDI_info(RDIConfig_Ntl, unsigned long *arg1, unsigned long *arg2)
```

This requests that details of the configuration block whose index (zero-based) is the word addressed by `arg1` should be returned to the `RDI_ConfigDesc` block addressed by `arg2`. Use this option only if `RDIInfo_Download` and `RDIConfig_Count` returned no errors.

RDIInfo_MemoryStats

```
int RDI_info(RDIInfo_MemoryStats, unsigned long *arg1,
             unsigned long *arg2)
```

This enquires whether the last four calls are available (`Memory_Access->Read_Clock`). The status returned is OK if they are available.

RDIPointStatus_Watch

```
int RDI_info(RDIPointStatus_Watch, unsigned long *arg1,
             unsigned long *arg2)
```

This can be used only if `RDIInfo_Points` sets bit 12 of `arg1`. When called with the handle of a watchpoint pointed to by `arg1`, this function returns the hardware resource number in the word pointed to by `arg1`, and the type of watchpoint in the word pointed to by `arg2`.

RDIPointStatus_Break

```
int RDI_info(RDIPointStatus_Break, unsigned long *arg1,
             unsigned long *arg2)
```

This is identical to `RDIPointStatus_Watch`, except that it is used for breakpoints.

RDISignal_Stop

```
int RDI_info(RDISignal_Stop, unsigned long *arg1, unsigned long *arg2)
```

This call requests that the debuggee stops execution.

RDISemiHosting_SetState

```
int RDI_info(RDISemiHosting_SetState, unsigned long *arg1)
```

This should be used only if `RDIInfo_SemiHosting` did not return an error. The setting of `arg1` is either:

0	disables semihosting
1	enables semihosting

RDISemiHosting_GetState

```
int RDI_info(RDISemiHosting_GetState, unsigned long *arg1)
```

This should be used only if `RDIInfo_SemiHosting` did not return an error. On return, `arg1` points to the current state of semihosting (0=off, 1= on).

RDISemiHosting_SetVector

```
int RDI_info(RDISemiHosting_SetVector, unsigned long *arg1)
```

This should be used only if `RDIInfo_SemiHosting` did not return an error. This sets the semihosting vector to be the value pointed to by `arg1`.

RDISemiHosting_GetVector

```
int RDI_info(RDISemiHosting_SetVector, unsigned long *arg1)
```

This should be used only if `RDIInfo_SemiHosting` did not return an error. On return, `arg1` points to the current value of the semihosting vector.

RDIICEBreaker_GetLocks

```
int RDI_info(RDIICEBreaker_GetLocks, unsigned long *arg1)
```

This should be used only if `RDIInfo_ICEBreaker` did not return an error. The value pointed to by `arg1` indicates which ICEBreaker breakpoints are locked.

RDIICEBreaker_SetLocks

```
int RDI_info(RDIICEBreaker_SetLocks, unsigned long *arg1)
```

This should be used only if `RDIInfo_ICEBreaker` did not return an error. The value pointed to by `arg1` indicates which ICEBreaker break points are locked.

RDIICEBreaker_GetLoadSize

```
int RDI_info(RDIICEBreaker_GetLoadsize, unsigned long *arg1)
```

This should be used only if `RDIInfo_Target` returned bit 12 set (Can Inquire Load Size). The maximum block size the Debug Agent can support is stored in the word pointed to by `arg1`.

RDICommsChannel_ToHost

```
int RDI_info(RDICommsChannel_ToHost, unsigned long *arg1,
             unsigned long *arg2)
```

This should be used only if the value returned by `RDIInfo_Target` had bit 16 (Debug Channel Exists) set. On entry, `arg1` points to a function `RDICCProc_ToHost`, and `arg2` contains `arg`. The type of `RDICCProc_ToHost` is:

```
void RDICCProc_ToHost(void *arg, ARMword data)
```

This should be called back to pass data from the target (via Debug Comms Channel) to the host. `arg` is the value passed in `arg2` by `RDICommsChannel_ToHost`.

RDICommsChannel_FromHost

```
int RDI_info(RDICommsChannel_FromHost, unsigned long *arg1,
             unsigned long *arg2)
```

This should be used only if the value returned by `RDIInfo_Target` had bit 16 (Debug Channel Exists) set. On entry, `arg1` points to a function `RDICCProc_FromHost`, and `arg2` contains `arg`.

The type of `RDICCProc_FromHost` is:

```
void RDICCProc_FromHost(void *arg, ARMword *data, int *valid)
```

This should be called back to request data from the host to be sent to the debuggee (via the Debug Comms Channel).

`arg` is the value passed in `arg2` by `RDICommsChannel_FromHost`.

`valid` indicates whether any data is available to be sent; if it is, it is stored at the word pointed to by `data`.

RDICycles

```
int RDI_info(RDICycles, unsigned long *arg1)
```

The debuggee returns, in the buffer addressed by `arg1`, the number of instructions and the number of S, N, I, C, and F cycles executed since it was initialized.

RDIErrorP

```
int RDI_info(RDIErrorP, unsigned long *arg1)
```

The debuggee returns, in the memory location addressed by `arg1`, the error pointer associated with the last return from `RDI_Execute` with status `RDIError_Error`.

RDISet_CmdLine

```
int RDI_info(RDISet_CmdLine, unsigned long *arg1)
```

Set the debuggee's command-line arguments before starting execution. `arg1` is a pointer to a NULL-terminated argument string, which must be no longer than 256 bytes, including the NULL.

RDISet_RDILevel

```
int RDI_info(RDISet_RDILevel, unsigned long *arg1)
```

Set the RDI/RDP protocol level to be used between the debugger and the debuggee. The level must be between the limits indicated by `RDIInfo_Target`.

RDISet_Thread

```
int RDI_info(RDISet_Thread, unsigned long *arg1)
```

Set the thread context for thread-sensitive functions such as breakpoint and watchpoint setting. The thread's handle must be passed in `arg1`.

RDI_DescribeCoPro

```
int RDI_info(RDI_DescribeCoPro, unsigned long *arg1,
             unsigned long *arg2)
```

This describes the registers of a coprocessor.

`arg1` points to the coprocessor number.

`arg2` points to a `Dbg_CoProDesc` block which the debuggee will fill in.

For full details of this structure see `Dbg_CoProDesc` in `dbg_cp.h`.

RDI_RequestCoProDesc

```
int RDI_info(RDI_RequestCoProDesc, unsigned long *arg1,
             unsigned long *arg2)
```

This function requests the description of the coprocessor.

`arg1` gives the number of the coprocessor

`arg2` points to a `Dbg_CoProDesc` block which the debuggee will fill in

For full details of this structure see `Dbg_CoProDesc` in `dbg_cp.h`.

RDIInfo_Log

```
int RDI_info(RDIInfo_TLog, unsigned long *arg1)
```

Return the RDI's logging state in the integer variable addressed by `arg1`. Bit 0 is set to log calls to RDI interfaces, and bit 1 to log RDP transactions.

RDIInfo_SetLog

```
int RDI_info(RDIInfo_SetLog, unsigned long *arg1)
```

Return the RDI's logging state to the integer value addressed by `arg1`. Bit 0 is set to log calls to RDI interfaces, and bit 1 to log RDP transactions.

RDIProfile_Stop

```
int RDI_info(RDIProfile_Stop, unsigned long *arg1)
```

This function specifies that profiling data should stop being collected. It should be used only if `RDIInfo_Target` returned bit 14 set in the value pointed to by `arg1` (to indicate that profiling is supported).

RDIProfile_Start

```
int RDI_info(RDIProfile_Start, unsigned long *arg1)
```

This call starts profiling. It should be used only if `RDIInfo_Target` returned bit 14 set in the value pointed to by `arg1` (to indicate that profiling is supported). `arg1` points to the interval in microseconds which should be used to start profiling.

RDIProfile_WriteMap

```
int RDI_info(RDIProfile_WriteMap, unsigned long *arg1)
```

This should be used only if `RDIInfo_Target` returned bit 14 set in the value pointed to by `arg1`. `arg1` points to an array of debuggee addresses. These addresses should be in increasing order, and are used to decide which count element in a corresponding array should be incremented when a value of the PC has been sampled. This works as follows:

`arg1[0]` = length of array
if
  PC lies between `arg1[i]` and `arg1[i+1]`
then
  `count[i]` should be incremented

RDIProfile_ReadMap

```
int RDI_info(RDIProfile_ReadMap, unsigned long *arg1,
             unsigned long *arg2)
```

This should be used only if `RDIInfo_Target` returned bit 14 set in the value pointed to by `arg1` (indicates that profiling is supported). On entry `arg1` points to the length of counts array to be read. `arg2` points to memory where the array of counts will be placed on exit. For more information, see `RDIProfile_WriteMap`.

RDIProfile_ClearCounts

```
int RDI_info(RDIProfile_ClearCounts, unsigned long *arg1)
```

This should be used only if `RDIInfo_Target` returned bit 14 set in the value pointed to by `arg1` (indicating that profiling is supported). All counts are reset to zero.

11.3 Error Codes

The symbolic values named here are defined in `rdi.h`.

`RDIError_NoError`	Everything worked
`RDIError_Reset`	Debuggee reset
`RDIError_UndefinedInstruction`	Tried to execute an undefined instruction
`RDIError_SoftwareInterrupt`	A SWI occurred
`RDIError_PrefetchAbort`	Execution ran into unmapped memory
`RDIError_DataAbort`	No memory at the specified address
`RDIError_AddressException`	Accessed > 26-bit address in 26-bit mode
`RDIError_IRQ`	An interrupt occurred
`RDIError_FIQ`	A fast interrupt occurred
`RDIError_Error`	A software error occurred
`RDIError_BranchThrough0`	Branch through location 0
`RDIError_NoMorePoints`	The last breakpoint/watchpoint
`RDIError_CantSetPoint`	Breakpoint/watchpoint resources exhausted
`RDIError_BreakpointReached`	Returned by `RDI_execute` and `RDI_step`
`RDIError_WatchpointAccessed`	Returned by `RDI_execute` and `RDI_step`
`RDIError_ProgramFinishedInStep`	End of the program reached while stepping
`RDIError_UserInterrupt`	You pressed Escape
`RDIError_NoSuchPoint`	Tried to clear a non-existant breakpoint/watchpoint
`RDIError_CantLoadConfig`	Configuration data could not be loaded
`RDIError_BadConfigData`	Configuration data was corrupted
`RDIError_NoSuchConfig`	Requested configuration has not been loaded
`RDIError_BufferFull`	Buffer became full during operation
`RDIError_OutOfStore`	Debug Agent ran out of memory
`RDIError_NotInDownLoad`	Illegal request made during DownLoad
`RDIError_PointInUse`	EmbeddedICE breakpoint is already in use
`RDIError_BadImageFormat`	Debug Agent could not make sense of AIF image
`RDIError_TargetRunning`	Target Processor did not stop (probably with EmbeddedICE system)
`RDIError_DeviceWouldNotOpen`	Failed to open serial/parallel port
`RDIError_NoSuchHandle`	No such memory description handle exists
`RDIError_ConflictingPoint`	Incompatible breakpoint already exists
`RDIError_SoftInitialiseError`	Recoverable error in RDI initialization. (You might need to use a different configuration.)

11.3.1 Information messages

The following messages pass information about the debuggee:

`LittleEndian`	The debuggee is little endian
`BigEndian`	The debuggee is big endian

11.3.2 Internal fault or limitation

The following errors indicate an internal fault or limitation:

`InsufficientPrivilege`	Supervisor state was not accessible to this debug monitor
`UnimplementedMessage`	Debuggee cannot honour this RDP request
`UndefinedMessage`	Corrupted RDP request
`IncompatibleRDILevel`	There is no common RDI level at which the debugger and debuggee can operate.

11.3.3 RDI errors

The following errors indicate misuse of the RDI or similar problems:

`NotInitialised`	`RDI_open` must be the first call
`UnableToInitialise`	The target world is broken
`WrongByteSex`	The debuggee cannot operate with the requested byte sex
`UnableToTerminate`	Target world was smashed by the debuggee
`BadInstruction`	It is illegal to execute this instruction
`IllegalInstruction`	The effect of executing this is undefined
`BadCPUStateSetting`	Tried to set the SPSR of user mode
`UnknownCoPro`	This co-processor is not connected
`UnknownCoProState`	Cannot execute this co-processor request
`BadCoProState`	Recognizably broken co-pro request
`BadPointType`	Misuse of the RDI
`UnimplementedType`	Misuse of the RDI
`BadPointSize`	Misuse of the RDI
`UnimplementedSize`	Halfwords are not yet implemented

11.4 Angel Debug Protocol (ADP)

The Angel Debug Protocol (ADP) is the communication protocol used between the ARM Debuggers and a remote debuggee. ADP is different from, and incompatible with, the ARM Remote Debug Protocol (RDP). ADP supercedes RDP.

Usually, the ADP is implemented on the host side as a module which translates RDI requests into ADP packets to be sent to the remote debuggee by a communications link of some sort.

The ADP gives the ARM Debuggers a uniform way to communicate with:

- a debug monitor running on ARM-based hardware accessed via a communications link, for example: Angel Debug Monitor running on an ARM PIE card, linked with serial, serial & parallel, or Ethernet
- a remote debug agent controlling an ARM processor via hardware debug support, for example: ARM's EmbeddedICE connected to the host via serial or serial and parallel, and the EmbeddedICE box connected to the debuggee via the JTAG port.

As the ADP evolves, new levels of specification are added and within any level of specification there are implementation options. This approach is taken so that a variety of minimal debug monitors and controlling debug agents can be accommodated without excessive overhead, and to ensure compatibility between debuggers and debug agents released at different times. As a result, a debugger using the ADP must negotiate its debug agent's capabilities and must not use capabilities its debug agent does not support.

11.4.1 The protocol stack

Debuggers and debug agents which communicate using ADP do so using not just ADP, but a stack of protocols which provide different capabilities.

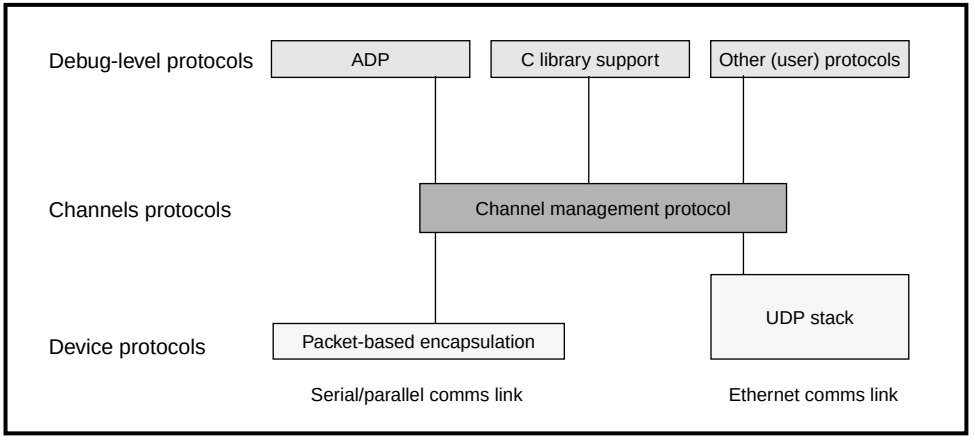


Figure 11-1: The protocol stack

The lowest level protocol(s) depend on the communications link being used. These are the device-dependent protocols.

Above this there is a channels layer protocol which allows a single communications link to support many sets of higher level protocols to go on simultaneously. Finally there is the ADP protocol, C Library Support protocol, and potentially other user-defined protocols.

11.4.2 Device-level protocols

The function of the device layer protocol is to encapsulate requests into packets and then transmit them across the communications link. This may involve many layers of protocol (eg. the Ethernet UDP stack), or a very thin layer such as for serial/parallel devices.

As an example, the device-level protocol for serial communication sends:

- a start-of-packet character
- the length of the packet
- some flags

It then escapes any characters in the data which match the start-of-packet, end-of-packet or escape character. After sending the data, a data checksum (CRC) is sent, and finally an end-of-packet character is sent.

On receipt of a packet, it is possible to detect a number of errors by checking for:

- no “out of packet” characters
- packets which are not the correct length
- packets which have a bad checksum

This layer passes on good and bad packets (indicating them as such) to the channels layer.

11.4.3 Channels-level protocol

This layer distinguishes the different sources of packets, by adding the Id of the channel. It is also responsible for checking to see if packets are missing, or are corrupt. In either case, this layer will request a resend.

The exact protocol used to accomplish this is provided in:

`hostchan.c`	on the host side
`channels.c`	on the target side

It does this by adding two packet-sequencing numbers (one for each end of the communications link), and checking that the sequencing numbers of any received packet are those it would expect. If not, resend requests are made as appropriate.

Note *Any one channel can support messages which are either host originated or target originated, but not both. This is described in **11.4.4 Debug-level protocols**.*

11.4.4 Debug-level protocols

Each channel can have a different protocol running on it at this level.

The major protocol is ADP, which supports debugging functionality between ARM debuggers and ARM Debug Agents/Monitors. For details of the exact protocol used, see the header file `adp.h`.

In brief, ADP runs over two channels:

- one for Host-originated requests
  (eg. read memory, write registers, set breakpoint, etc.)
- one for Target-originated messages (gives reasons why execution on the target has stopped; for example, breakpoint, exception, etc.)

Each message contains:

1. A reason code.
2. Multi-threaded debug support fields (currently unused).
3. Data, as indicated by the reason code. There may also be a sub-reason code which further specifies some requests. The data will always include a status field to indicate success or otherwise of the request.

Replies are of the same format.

C library support protocol

The C Library support protocol runs on a single channel since it is always target-originated. The messages are the same structure as those for the ADP. For more details, see the header file `sys.h`.

File Format Reference

12 ARM Image Format

This section describes the file formats used by the ARM Software Development Toolkit.

12.1 Overview of ARM Image Format

ARM Image Format (AIF) is a simple format for ARM executable images, consisting of:

- a 128-byte header
- the image's code
- the image's initialized static data

At its most sophisticated, AIF can be considered to be a collection of envelopes which enwrap a plain binary image, as follows:

- The outer wrapper allows the inner layers to be compressed using any compression algorithm that supports efficient decompression at image load time, either by the loader or by the loaded image itself. In particular, AIF defines a simple structure for images which decompress themselves, consisting of:
  - AIF header
  - compressed contents
  - decompression tables
  - decompression code
- The next layer of wrapping deals with relocating the image to its load address. Three options are supported:
  - link-time relocation
  - load-time relocation to the address where the image has been loaded
  - load-time relocation to a fixed offset from the top of memory

  In particular, an AIF image is capable of self-relocation or self-location (to the high-address end of memory), followed by self-relocation. Once an AIF image has been decompressed and relocated, it can create its own zero-initialized area.
- Finally, the enwrapped image is entered at the (unique) entry point found by the linker in the set of input object modules.

12.2 AIF Flavors

Three flavors of AIF are supported:

Executable AIF

This 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, setting to zero its own zero-initialized data, etc. An executable AIF image is loaded at its load address (which may be arbitrary if the image is relocatable), and entered at the same address. Eventually, control passes to a branch to the image's entry point.

The header is part of the image itself. This variant can be executed by entering the header at its first word. 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

The header is not part of the image, but merely describes it. This variant is intended to be loaded by a program which interprets the header, and prepares the image following it for execution.

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.

Non-executable AIF must be processed by an image loader, which loads the image at its load address and prepares it for execution as detailed in the AIF header. The header is then discarded.

Extended AIF

This is not directly executable. It contains a scatter-loaded image. It has an AIF header which points to a chain of load regions within the file. The image loader should place these regions at the correct place in memory.

12.2.1 Executable AIF

It is assumed that on entry to a program in AIF, the general registers contain nothing of value to the program (the program is expected to communicate with its operating environment using SWI instructions or by calling functions at known, fixed addresses).

A program image in AIF is loaded into memory at its load address, and entered at its first word. The load address may be:

- an implicit property of the type of the file containing the image (as is usual with UNIX executable file types, etc.)
- read by the program loader from offset 0x28 in the file containing the AIF image
- given by some other means; for example, by instructing an operating system or debugger to load the image at a specified address in memory

12.2.2 Compressed images

An AIF image may be compressed and can be self-decompressing (to support faster loading from slow peripherals, and better use of space in ROMs and delivery media such as floppy disks). An AIF image is compressed by a separate utility which adds self-decompression code and data tables to it.

12.2.3 Relocation

If created with appropriate linker options, an AIF image may relocate itself at load time. Two kinds of self-relocation are supported:

- relocate to load address (the image can be loaded anywhere and will execute where loaded)
- self-move up memory, leaving a fixed amount of workspace above, and relocate to this address (the image is loaded at a low address and will move to the highest address which leaves the required workspace free before executing there)

The second kind of self-relocation can only be used if the target system supports an operating system or monitor call which returns the address of the top of available memory.

The ARM linker provides a simple mechanism for using a modified version of the self-move code illustrated in ***12.4 Zero-initialization code*** on page 12-10, allowing AIF to be tailored easily to new environments.

12.2.4 Debugging

AIF images support being debugged by the ARM symbolic debugger. Low-level and source-level support are orthogonal, and both, either, or neither kind of debugging support need be present in an AIF image. For details of the format of the debugging tables see ***Chapter 15, ARM Symbolic Debug Table Format***.

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.

12.2.5 AIF output

In order to produce an AIF output there must be:

- no unresolved symbolic references between the input objects (each reference must resolve directly or via an input library)
- exactly one input object containing a program entry point (or no input area containing an entry point, and the entry point given using an `-Entry` option)
- either an absolute load address or the relocatable option given to the linker (the self-location option is system-dependent)

12.3 The Layout of AIF

12.3.1 Compressed AIF image

The layout of a compressed AIF image is as follows:

1. Header
2. Compressed image
3. Decompression data (position-independent)
4. Decompression code (position-independent)

The header described below is small and fixed in size. In a compressed AIF image, the header is *not* compressed.

12.3.2 Uncompressed AIF image

An uncompressed image has the following layout:

1. Header
2. Read-only area
3. Read-write area
4. Debugging data (optional)
5. Self-relocation code (position-independent)
6. 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.

12.3.3 Debugging

Debugging data is absent unless the image has been linked using the linker's `-d` option and, in the case of source-level debugging, unless the components of the image have been compiled using the compiler's `-g` option.

After the execution of the self-relocation code (or if the image is not self-relocating) the image has the following layout:

1. Header
2. Read-only area
3. Read-write area
4. Debugging data (optional)

At this stage, a debugger is expected to copy any debugging data to somewhere safe, otherwise it will be overwritten by the zero-initialized data and/or the heap/stack data of the program. A debugger can take control at the appropriate moment by copying, then modifying the third word of the AIF header (see ***Figure 12-1: AIF header layout***).

00:	BL DecompressCode	NOP if the image is not compressed.	Note 1
04:	BL SelfRelocCode	NOP if the image is not self-relocating.	
08:	BL DBGInit/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.	Note 2
10:	Program Exit Instr	...last attempt in case of return.	Note 3
14:	Image ReadOnly size	Includes header size if executable AIF; excludes header size if non-executable AIF.	Note 4
18:	Image ReadWrite size	Exact size (a multiple of 4 bytes).	
1C:	Image Debug size	Exact size (a multiple of 4 bytes).	
20:	Image zero-init size	Exact size (a multiple of 4 bytes).	
24:	Image debug type	0, 1, 2, or 3 .	Note 6
28:	Image base	Address where the image (code) was linked.	
2C:	Work space	Minimum work space (in bytes) to be reserved by a self-moving relocatable image.	
30:	Address mode: 26/32 + 3 flag bytes	LS byte contains 26 or 32; bit 8 set when using a separate data base.	Note 7
34:	Data base	Address where the image data was linked.	
38:	Two reserved words (initially 0)		Note 8
40:	Debug Init Instr	NOP if unused.	Note 9
44:	Zero-init code (15 words as below)	Header is 32 words long.	

Figure 12-1: AIF header layout

Notes

1 `NOP` is encoded as `MOV r0, r0`.

2 `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. Care is taken to ensure that the instruction sequences which compute addresses from these r14 values work in both 26-bit and 32-bit ARM modes.

3 `Program Exit Instruction` will usually be a SWI causing program termination. On systems which lack this, 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.

4 `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).

5 An AIF image is restartable if, and only if, the program it contains is restartable (an AIF image is *not* re-entrant). If an AIF image is to be restarted following its decompression, the first word of the header must be set to `NOP`. Similarly, 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 segment; both decompression and relocation code must write to it. On systems with memory protection, both the decompression code and 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).

6 `Image debug type` has the following meaning:

- 0 No debugging data are present.
- 1 Low-level debugging data are present.
- 2 Source level (ASD) debugging data are present.
- 3 1 and 2 are present together.

All other values of image debug type are reserved to ARM Ltd.

7 `Address mode` 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
- 32 indicates that the image was linked for a 32-bit ARM mode, and may not execute correctly in a 26-bit mode

(0 indicates an old-style 26-bit AIF header)

If the Address mode word has bit 8 set ((*`address_mode`* `& 0x100) != 0`), the image was linked with separate code and data bases (usually the data is placed immediately after the code). Here, the word at offset 0x34 contains the base address of the image's data.

8 `FAT AIF images.` In these 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.

9 `Debug Initialization Instruction` (if used) is expected to be a SWI instruction which alerts a resident debugger that a debuggable image is starting to execute. The ARM cross-linker sets this field to `NOP` by default, but you can customize it by providing your own template for the AIF header in an area called `AIF_HDR` in the *first* object file in the input list to armlink.

12.4 Zero-initialization code

The zero-initialization code is as follows:

```
        NOP                 ; or Debug Init Instruction
        SUB    r12,r14,pc   ; base+12+[PSR]-(ZeroInit+12+[PSR])
                            ; = base-ZeroInit
        ADD    r12,pc,r12   ; base-ZeroInit+ZeroInit+16
                            ; = base+16
        LDMIB  r12,{r0-r3}  ; various sizes
        SUB    r12,r12,#0x10; image base
        LDR    r2,[r12,#0x30]
        TST    r2,#0x100
        LDRNE  r12,[r12,#0x34]
        ADDEQ  r12,r12,r0   ; + rO size
        ADD    r12,r12,r1   ; + RW size
                            ; = base of 0-init area
        MOV    r0,#0
        CMP    r3,#0
00      MOVLE  pc,r14       ; nothing left to do
        STR    r0,[r12],#4
        SUBS   r3,r3,#4
        B      %B00
```

12.4.1 Self-move and self-relocation code

This code is added to the end of an AIF image by the linker, immediately before the list of relocations (which is terminated by `-1`).

Note *The code is entered via a* `BL` *from the second word of the AIF header so, on entry, r14 -> AIFHeader + 8. In 26-bit ARM modes, r14 also contains a copy of the PSR flags.*

On entry, the relocation code calculates the address of the AIF header (in a CPU-independent manner) and decides whether the image needs to be moved. If the image does not need to be moved, the code branches to `RelocateOnly`.

```
RelocCode
    NOP                     ; required by ensure_byte_order()
                            ; and used below.
    SUB    ip, lr, pc       ; base+8+[PSR]-(RelocCode+12+[PSR])
                            ; = base-4-RelocCode
    ADD    ip, pc, ip       ; base-4-RelocCode+RelocCode+16 = base+12
    SUB    ip, ip, #12      ; -> header address
    LDR    r0, RelocCode    ; NOP
    STR    r0, [ip, #4]     ; won't be called again on image re-entry
    LDR    r9, [ip, #&2C]   ; min free space requirement
    CMPS   r9, #0           ; 0 => no move, just relocate
   BEQ    RelocateOnly
```

If the image needs to be moved up memory, the top of memory has to be found. Here, a system service (SWI 0x10) is called to return the address of the top of memory in r1.

Note *This is system-specific and should be replaced by whatever code sequence is appropriate to the environment.*

```
    LDR     r0, [ip, #&20]  ; image zero-init size
    ADD     r9, r9, r0      ; space to leave =
                            ; min free + zero init
    SWI     #&10            ; return top of memory in r1.
```

The following code calculates the length of the image inclusive of its relocation data, and decides whether a move up store is possible.

```
    ADR     r2, End         ; -> End
01  LDR     r0, [r2], #4    ; load relocation offset,
                            ; increment r2
    CMNS    r0, #1          ; terminator?
    BNE     %B01            ; No, so loop again
    SUB     r3, r1, r9      ; MemLimit - freeSpace
    SUBS    r0, r3, r2      ; amount to move by
    BLE     RelocateOnly    ; not enough space to move...
    BIC     r0, r0, #15     ; a multiple of 16...
    ADD     r3, r2, r0      ; End + shift
    ADR     r8, %F02        ; intermediate limit for
                            ; copy-up
```

Finally, the image copies itself four words at a time, taking care over the direction of copy, and jumping to the copied code as soon as it has copied itself.

```
02  LDMDB   r2!, {r4-r7}
    STMDB   r3!, {r4-r7}
    CMPS    r2, r8          ; copied the copy loop?
    BGT     %B02            ; not yet
    ADD     r4, pc, r0
    MOV     pc, r4          ; jump to copied copy code
03  LDMDB   r2!, {r4-r7}
    STMDB   r3!, {r4-r7}
    CMPS    r2, ip          ; copied everything?
    BGT     %B03            ; not yet
    ADD     ip, ip, r0      ; load address of code
    ADD     lr, lr, r0      ; relocated return address
```

Whether the image has moved itself or not, control eventually arrives here, where the list of locations to be relocated is processed. Each location is word sized and is relocated by the difference between the address where the image was loaded (the address of the AIF header) and the address where the image was linked (stored at offset 0x28 in the AIF header):

```
RelocateOnly
    LDR    r1, [ip, #&28]  ; header + 0x28 = code base
                           ; set by Link
    SUBS   r1, ip, r1      ; relocation offset
    MOVEQ  pc, lr          ; relocate by 0 so nothing to do
    STR    ip, [ip, #&28]  ; new image base = actual load address
    ADR    r2, End         ; start of reloc list
04  LDR    r0, [r2], #4    ; offset of word to relocate
    CMNS   r0, #1          ; terminator?
    MOVEQ  pc, lr          ; yes => return
    LDR    r3, [ip, r0]    ; word to relocate
    ADD    r3, r3, r1      ; relocate it
    STR    r3, [ip, r0]    ; store it back
    B      %B04            ; and do the next one
End                        ; The list of offsets of locations to
                           ; relocate starts here,
                           ; terminated by -1.
```

You can customize the self-relocation and self-moving code generated by the linker by providing your version of it in an area called AIF_RELOC in the *first* object file in the linker's input list.

13 ARM Object Library Format

This section describes the file formats used by the ARM Software Development Toolkit.

13.1 Overview of ARM Object Library Format

This section defines a file format called *ARM Object Library Format*, or *ALF*, which 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. For a description of the Chunk File Format, see ***14.2.1 Chunk file format*** on page 14-5.

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

These chunks are described in detail in ***Chapter 15, ARM Symbolic Debug Table Format***.

13.1.1 Terminology

The terminology in this chapter is as follows:

byte	means 8 bits, considered unsigned unless otherwise stated, usually used to store flag bits or characters
halfword	means 16 bits, or 2 bytes, usually considered unsigned
word	means 32 bits, or 4 bytes, usually considered unsigned
string	means a sequence of bytes terminated by a NUL (0x00) byte The NUL byte is part of the string but is not counted in the string's length.

13.2 Endianness and Alignment

There are two sorts of ALF: *little-endian* and *big-endian:*

Little-endian The least significant byte of a word or halfword has the lowest address of any byte in the (half-)word. This *byte sex* is used by DEC and Intel amongst others.

Big-endian The most significant byte of a (half)word has the lowest address. This byte sex is used by IBM, Motorola and Apple, amongst others.

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.

13.2.1 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.

13.3 Library File Format

For library files, the first part of each chunk's 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, named as follows:

Chunk	Chunk name
Directory	LIB_DIRY
Time stamp	LIB_TIME
Version	LIB_VSRN
Data	LIB_DATA
Symbol table	OFL_SYMT object code libraries only
Time stamp	OFL_TIME object code libraries only

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.

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 NUL (0) bytes; applications which read LIB_DIRY or OFL_SYMT entries should make no assumptions about the values of padding bytes beyond the first, string-terminating NUL byte.

13.3.1 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.

Pictorially:

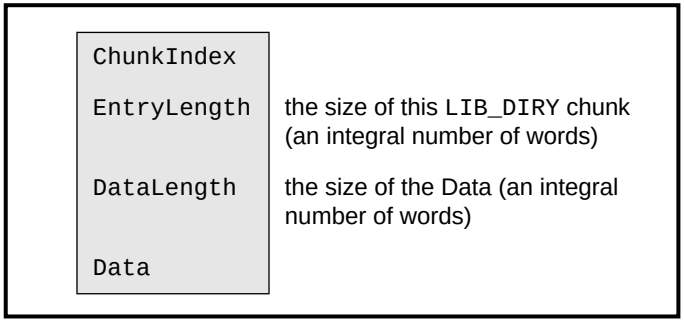


Figure 13-1: The LIB_DIRY chunk

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_VSRN`, 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 ***13.4 Time Stamps*** on page 13-6. Its value is an encoded version of the last-modified time of the file from which the library member was created.

13.3.2 LIB_VSRN

The version chunk contains a single word whose value is 1.

13.3.3 LIB_DATA

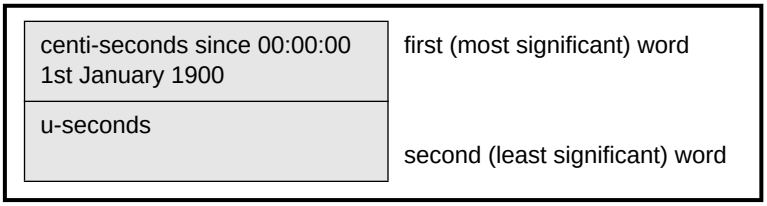
A LIB_DATA chunk contains one of the library members indexed by the LIB_DIRY chunk. The endianness or byte order of this data is, by assumption, the same as the byte order of the containing library/chunk file.

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.

13.4 Time Stamps

A library time stamp is a pair of words encoding the following:

- a six-byte count of centi-seconds since the start of the 20th century
- a two-byte count of microseconds since the last centi-second (usually 0).



The first word stores the most significant four bytes of the six-byte count; the least significant two bytes of the count are in the most significant half of the second word.

The least significant half of the second word contains the microsecond count; it is usually 0.

Time stamp words are stored in target system byte order; they must have the same endianness as the containing chunk file.

13.4.1 LIB_TIME

The LIB_TIME chunk contains a two-word (eight-byte) time stamp recording when the library was last modified.

13.5 Object Code Libraries

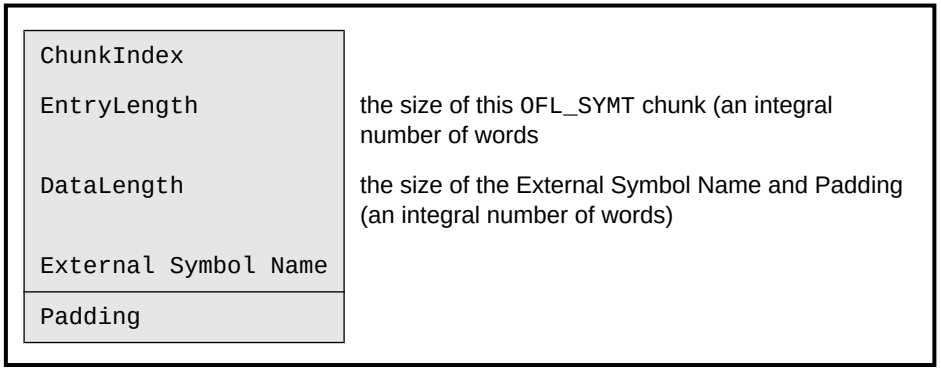
An object code library is a library file whose members are files in ARM Object Format. An object code library contains two additional chunks:

- an external symbol table chunk named `OFL_SYMT`
- a time stamp chunk named `OFL_TIME`

13.5.1 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 NUL padding, as follows:



`OFL_SYMT` entries do not contain time stamps.

13.5.2 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 ***13.4 Time Stamps*** on page 13-6).

14 ARM Object Format

This section defines a file format called *ARM Object Format* (AOF) which is used by language processors for ARM-based systems.

14.1 ARM Object Format

AOF is a simple object format, similar in complexity and expressive power to the UNIX `a.out` format. It provides a generalized superset of `a.out`'s descriptive facilities. AOF is designed to be simple to generate and to process, rather than to be expressive or compact.

ARM Object Format directly supports the ARM Procedure Call Standard (APCS), which is described in ***Chapter 5, ARM Procedure Call Standard***.

In this section:

object file	refers to a file in ARM Object Format
linker	refers to the ARM linker
byte	is 8 bits long, and is considered unsigned unless otherwise stated. This is usually used to store flag bits or characters
halfword	is 16 bits or 2 bytes, and is usually considered unsigned
word	is 32 bits or 4 bytes, and is usually considered unsigned
string	is a sequence of bytes terminated by a NUL (0x00) byte. The NUL byte is part of the string but is not counted in the string's length
address	for data in a file, this means *offset from the start of the file*

14.1.1 Areas

An object file written in AOF consists of any number of named, attributed *areas*. Attributes include:

- read-only
- re-entrant
- code
- data
- position-independent etc.

For details see ***14.4 Attributes and Alignment*** on page 14-11).

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 re-entrant code uses a separate based area for address constants).

14.1.2 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

14.1.3 AOF and the linker

The ARM linker accepts input files in AOF format and can generate output in the same format, as well as in a variety of image formats. The ARM linker is described in ***Chapter 3, Linker***. When AOF is used as an output format, the linker does the following with its input object files:

- merges similarly named and attributed areas
- performs PC-relative relocations between merged areas
- rewrites symbol-relative relocation directives between merged areas, as area based relocation directives belonging to the merged area
- minimizes the symbol table

Unresolved references remain unresolved, and the output AOF file may be used as the input to a further link step.

14.1.4 Byte sex or endianness

There are two types of AOF:

Little-endian
: The least significant byte of a word or halfword has the lowest address of any byte in the (half)word. This *byte sex* is used by DEC and Intel, amongst others.

Big-endian
: The most significant byte of a (half)word has the lowest address. This byte sex is used by IBM, Motorola and Apple, amongst others.

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).

The two sorts of AOF cannot 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.

14.1.5 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.

14.2 Overall Structure of an AOF File

An AOF file contains a number of separate but related 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.

14.2.1 Chunk file format

The object file format defines five chunks:

- AOF header
- AOF areas
- producer's identification
- symbol table
- string table

These are described ***14.2.2 ARM object format*** on page 14-6.

A chunk is accessed via a header at the start of the file. 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 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. Pictorially, the layout of a chunk file is as follows:

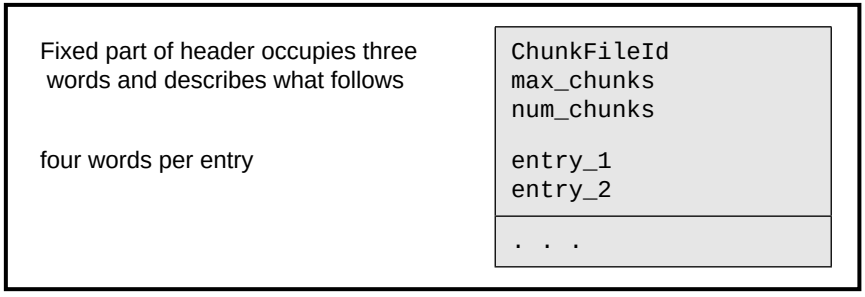


Figure 14-1: Chunk file layout

`ChunkFileId` marks the file as a chunk file. Its value is 0xC3CBC6C5. The endianness of the chunk file can be deduced 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 `maxChunks`. It is redundant in that it can be found by scanning the entries.

Each entry in the chunk file header consists of four words in order:

`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 NUL-terminated string, independent of endianness.

For AOF files, the first part of each chunk's name is `OBJ_`; the second components are defined in ***14.2.2 ARM object format*** below.

14.2.2 ARM object format

Each piece of an object file is stored in a separate, identifiable chunk. AOF defines five chunks as follows:

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 (see ***14.8 The String Table Chunk (OBJ_STRT)*** on page 14-21). 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's header.*

14.3 Format of the AOF Header Chunk

The AOF header consists of two parts, which appear contiguously in the header chunk:

part 1 is of fixed size and describes the contents and nature of the object file

part 2 has a variable length (specified in the fixed part of the header), and consists of a sequence of *area declarations* describing the code and data areas within the `OBJ_AREA` chunk

Pictorially, the AOF header chunk has the format shown in ***Figure 14-2: AOF header chunk*** on page 14-9.

`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 version of AOF with which the object file complies:

- version 1.50 is denoted by decimal 150
- version 2.00 is denoted by 200
- this version is denoted by decimal 310 (0x136)

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 below). 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 which follow the fixed part of the AOF header.

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`, 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. The entry address is defined to be the base address of this area plus `Entry Offset`.

A value of 0 for `Entry Area Index` signifies that no program entry address is defined by this AOF file.

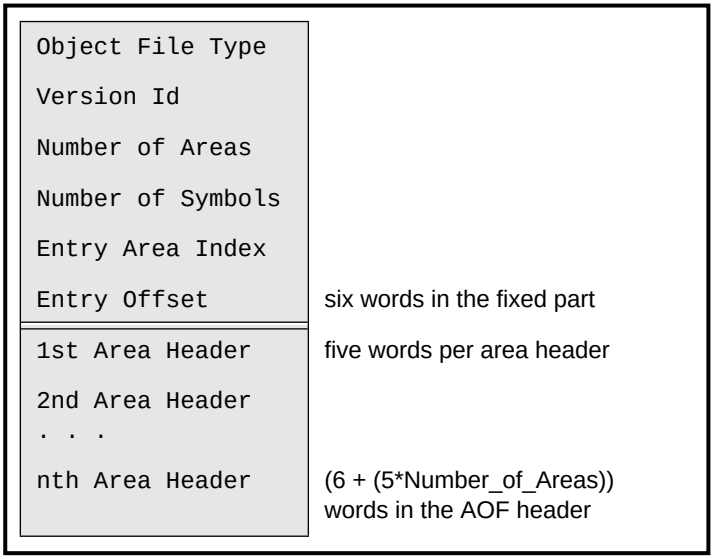


Figure 14-2: AOF header chunk

14.3.1 Format of area headers

The area headers follow the fixed part of the AOF header. Each area header has the following format:

`Area name`	(offset into string table)
`Attributes + Alignment`	
`Area Size`	
`Number of Relocations`	
`Base Address or 0`	(five words in total)

Each area within an object file must be given a unique name.

`Area Name` gives the offset of that name in the string table (stored in the `OBJ_STRT` chunk; see ***14.8 The String Table Chunk (OBJ_STRT)*** on page 14-21).

`Area Size` gives the size of the area in bytes, which must be a multiple of 4. Unless the `Not Initialised bit` (bit 12) is set in the area attributes (see ***14.4 Attributes and Alignment***), 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 which apply to this area (which is equivalent to the number of relocation records following the area's contents in the OBJ_AREA chunk; see ***14.5 Format of the AREAS Chunk*** on page 14-14).
`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.

14.4 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 $2^{alignment}$).

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. For details see ***3.4 Area Placement and Sorting Rules*** on page 3-13.

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:

set	indicates code in the area.
unset	indicates data in the area.

Bit 10 specifies that the area is a common definition.

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.

Note *If both bits 10 and 11 are set, bit 11 is ignored.*

Common areas

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 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 (eg. 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. For details, refer to ***5.3.1 32-bit PC vs 26-bit PC*** on page 5-11. Such code may be incompatible with code that complies with a 26-bit variant of the APCS.

Bit 17 encodes the *re-entrant* attribute, and denotes that code in this area complies with a re-entrant variant of the APCS.

Bit 18 when set, denotes that code in this area uses the ARM's extended floating-point instruction set. Specifically, function entry and exit use the LFM and SFM floating-point save and restore instructions rather than multiple LDFEs and STFEs. 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. Such code may be incompatible with code that complies with a limit-checked variant of the APCS.

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 *Software Development Toolkit User Guide*.

Bits 20–27 encode additional attributes of data areas, and must be non-zero only if the area does not have the code attribute (bit 9) unset.
Bits 20–22 can be non-zero for code areas.

Bit 20 encodes the *based* attribute, denoting that the area is addressed via link-time-fixed offsets from a base register (encoded in bits 24-27). Based areas have a special role in the construction of shared libraries and ROM-able code, and are treated specially by the linker (refer to ***3.6.6 Based area relocation*** on page 3-17).

Bit 21 encodes the *Shared Library Stub Data* attribute. In a link step involving layered shared libraries, there may be several copies of the stub data for any library not at the top level. In other respects, areas with this attribute are treated like data areas with the *common definition* (bit 10) attribute. Areas which also have the *zero-initialized* attribute (bit 12) are treated much the same as areas with the *common reference* (bit 11) attribute. This attribute is not usually set by language processors, but is set only by the linker.

Bits 22–23 are reserved and must be set to 0.

Bits 24–27 encodes the base register used to address a *based* area. If the area does not have the *based* attribute, these bits are set to 0.

Bits 28–31 are reserved and must be set to 0.

14.4.1 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	Re-entrant code
18	0x00040000	Uses extended FP inst set
19	0x00080000	No software stack checking
20	0x00100000	Thumb Code area
21	0x00200000	Area may contain ARM halfword instructions
22	0x00400000	Area suitable for ARM/Thumb interworking
		(Data areas only)
20	0x00100000	Based area
21	0x00200000	Shared library stub data
24–27	0x0F000000	Base register for based area

Table 14-1: Area attributes

14.5 Format of the AREAS Chunk

The areas chunk (OBJ_AREA) contains the actual area contents (code, data, debugging data, etc.) together with their associated relocation data. Its *chunkId* is OBJ_AREA. An area is simply a sequence of byte values. The endianness of the words and halfwords within it must agree with that of the containing AOF file. Pictorially, an area's 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.

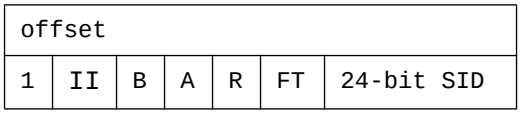
14.6 Relocation Directives

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.

Pictorially, a relocation directive looks like:



`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):

Value	Description
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.
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:

Value	Description
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

Thumb 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:

Value	Description
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.

14.6.1 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 C:

```
subject_field = subject_field + relocation_value
```

14.6.2 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 C:

```
subject_field = subject_field + (relocation_value -
                    base_of_area_containing(subject_field))
```

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.

Note *This allows compilers to perform the tail-call optimization on re-entrant code. For details, refer to* ***3.6.4 Forcing use of an inter-link-unit entry point*** *on page 3-16.*

14.6.3 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 C:

```
subject_field = subject_field + (relocation_value
   - base_of_area_group_containing(relocation_value))
```

For example, when generating re-entrant code, the C compiler places address constants in an adcon area based on register sb, and loads them using sb relative LDRs. At link time, separate adcon 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. For further details, refer to ***3.6.6 Based area relocation*** on page 3-17.

Bits 29 and 30 of the relocation flags word must be 0; bit 31 must be 1.

14.7 Symbol Table Chunk Format (OBJ_SYMT)

The NumberofSymbols 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 has this format:

```
Name
Attributes
Value
Area Name        4 words per entry
```

where:

`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).

`Name` is the offset in the string table (in chunk `OBJ_STRT`) of the character string name of the symbol.

`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.

14.7.1 Symbol attributes

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's 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 encodes the *strong* attribute which is meaningful only if the symbol is an external defining occurrence (bits 1, 0 = 11). In turn, this attribute only has meaning if there is a non-strong, external definition of the same symbol in another object file. In this case, references to the symbol from outside of the file containing the strong definition resolve to the strong definition, while those within the file containing the strong definition resolve to the non-strong definition.

This attribute allows a kind of link-time indirection to be enforced. Usually, a strong definition will be absolute, and will be used to implement an operating system's entry vector having the *forever binary* property.

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 the *simple leaf function* attribute which is meaningful only if this symbol defines the entry point of a sufficiently simple leaf function (a leaf function is one which calls no other function). For a re-entrant leaf function, it denotes that the function's inter-link-unit entry point is the same as its intra-link-unit entry point. For details of the significance of this attribute to the linker, refer to ***3.6.4 Forcing use of an inter-link-unit entry point*** on page 3-16.

Thumb Bit 12 is the Thumb attribute, which is set if the symbol is a Thumb symbol.

14.7.2 Symbol attribute summary

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
5	0x00000020	Strong attribute
6	0x00000040	Common attribute
		Code symbols only:
8	0x00000100	Code area datum attribute
9	0x00000200	FP args in FP regs attribute
11	0x00000800	Simple leaf function attribute
12	0x00001000	Thumb symbol

Table 14-2: Symbol attributes

14.8 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 NUL (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.

14.9 The Identification Chunk (OBJ_IDFN)

This chunk should contain a string of printable characters (codes 10–13 and 32–126) terminated by a NUL (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.

15 ARM Symbolic Debug Table Format

This section specifies the format of symbolic debugging data generated by ARM compilers, which is used by the ARM symbolic debugger to support high-level language-oriented, interactive debugging.

ARM Symbolic Debug Table Format

15.1 Overview of ARM Symbolic Debug Table Format

For each separate compilation unit (called a *section*) the compiler produces debugging data, and a special *area* in the object code (see ***Chapter 14, ARM Object Format*** for an explanation of ARM Object Format, including areas and their attributes). Debugging data are position-independent, containing only relative references to other debugging data within the same section, and relocatable references to other compiler-generated areas.

Debugging data areas are combined by the ARM linker into a single contiguous section of a program image. For details of the ARM linker's capabilities see ***Chapter 3, Linker***. For a description of the linker's principal output format see ***Chapter 12, ARM Image Format***.

The format of debugging data allows for a variable amount of detail. This potentially allows you to trade off between memory used, disk space used, execution time, and debugging detail.

Assembly-language-level debugging is also supported, though in this case the debugging tables are generated by the linker. If required, the assembler can generate debugging table entries relating code addresses to source lines. Low-level debugging tables appear in an extra section item, as if generated by an independent compilation.

Low-level and high-level debugging are orthogonal facilities, though the debugger allows you to move smoothly between levels if both sets of debugging data are present in an image.

15.1.1 Terminology

The terms *byte, word,* and *halfword* are used to mean:

byte	8 bits, usually considered unsigned
word	32 bits (4 bytes), often considered signed
halfword	16 bits (2 bytes), also called a *short* Halfwords are unused, except in the long form of *LineInfo* items.

15.2 Order of Debugging Data

A debug data area consists of a series of *items*. The arrangement of these items mimics the structure of the high-level language program itself.

For each debug area, the first item is a *section* item, giving global information about the compilation, including a code identifying the language, and flags indicating the amount of detail included in the debugging tables.

Each definition datum, function, procedure, etc., in the source program has a corresponding debug data item, which appear in an order corresponding to the order of definitions in the source. This means that nested structures in the source program are preserved in the debugging data, and the debugger can use these structure to make deductions about the scope of various source-level objects. For procedure definitions, there are two debug items:

procedure item	marks the definition itself
endproc item	marks the end of the procedure's body and the end of any nested definitions

If procedure definitions are nested, so are the *procedure endproc* brackets. Variable and type definitions made at the outermost level appear outside of all procedure/endproc items.

Information about the relationship between the executable code and source files is collected together and appears as a *fileinfo* item, which is always the final item in a debugging area. Because of the C language's #include facility, the executable code produced from an outer-level source file may be separated into disjoint pieces interspersed with that produced from the included files. Therefore, source files are considered to be collections of *fragments*, each corresponding to a contiguous area of executable code, and the *fileinfo* item is a list with an entry for each file, each in turn containing a list with an entry for each fragment. The fileinfo field in the *section* item addresses the *fileinfo* item itself. In each *procedure* item there is a *fileentry* field, which refers to the file-list entry for the source file containing the procedure's start; there is a separate one in the *endproc* item because it may possibly not be in the same source file.

15.2.1 Endianness and the encoding of debugging data

The ARM can be configured to use either a little-endian memory system (least significant byte of each four-byte word has the lowest address), or a big-endian memory system (most significant byte of each four-byte word has the lowest address).

In general, the code to be generated varies according to the byte sex (or endianness) of the target, and the linker has insufficient information to change the byte sex of an object file. Therefore, object files are encoded using the byte order of the intended target, independently of the byte order of the host system on which the compiler or assembler runs. The ARM linker accepts inputs having either byte order, but rejects mixed sex inputs, and generates its output using the same byte order. This means that producers of debugging tables must be prepared to generate them in either byte order, as required. In turn, this requires definitions to be very clear about when a four-byte word is being used (which will

require reversal on output or input when cross-sex compiling or debugging), and when a sequence of bytes is being used (which requires no special treatment provided it is written and read as a sequence of bytes in address order).

15.3 Representation of Data Types

Several of the debugging data items (eg. procedure and variable) have a *type* word field to identify their data type. This field contains:

- in the most significant 24 bits, a code to identify a base type
- in the least significant 8 bits, a pointer count:

  0 denotes the type itself

  1 denotes a pointer to the type

  2 denotes a pointer to a pointer

For simple types the code is a positive integer as follows (all codes are decimal):

```
    void                0
signed integers
    single byte         10
    halfword            11
    word                12
    double word         13
unsigned integers
    single byte         20
    halfword            21
    word                22
    double word         23
floating point
    float               30
    double              31
    long double         32
complex
    single complex      41
    double complex      42
functions
    function            100
```

For compound types (arrays, structures, etc.) there is a special kind of debug data item (array, struct, etc.) to give details such as array bounds and field types. The type code for compound types is negative; the negation of the (byte) offset of the debug item from the start of the debugging area.

Set types in Pascal are not treated in detail: the only information recorded for them is the total size occupied by the object in bytes. Neither are Pascal *file* variables supported by the debugger, since their behavior under debugger control is unlikely to be helpful to you.

Fortran character types are supported by special kinds of debugging data item, the format of which is specific to each Fortran compiler.

15.3.1 Representation of source file positions

Several of the debugging data items have a *sourcepos* field to identify a position in the source file. This field contains a line number and character position within the line packed into a single word. The most significant 10 bits encode the character offset (zero-based) from the start of the line and the least significant 22 bits give the line number.

15.3.2 The code and length field

The first word of each debugging data item contains the byte length of the item (encoded in the most significant 16 bits), and a code identifying the kind of item (in the least significant 16 bits). The defined codes are:

1	section
2	procedure/function definition
3	endproc
4	variable
5	type
6	struct
7	array
8	subrange
9	set
10	fileinfo
11	contiguous enumeration
12	discontiguous enumeration
13	procedure/function declaration
14	begin naming scope
15	end naming scope
16	bitfield
17	macro definition
18	macro undefinition
19	class
20	union
32	FP map fragment

The meaning of the second and subsequent words of each item is defined in the following sections.

If a debugger encounters a code it does not recognize, it uses the length field to skip the item entirely. This discipline allows the debugging tables to be extended without invalidating existing debuggers.

15.3.3 Text names in items

Where items include a string field, the string is packed into successive bytes beginning with a length byte, and padded at the end to a word boundary with 0 bytes. The length of a string is in the range [0..255] bytes.

15.3.4 Offsets in file and addresses in memory

When an item contains a field giving an offset in the debugging data area (usually to address another item), this means a byte offset from the start of the debugging data for the whole section (in other words, from the start of the *section* item).

15.4 Section Items

A section item is the first item of each section of the debugging data. After its code and length word it contains the fields listed below. First, there are four flag bytes:

`lang` a byte identifying the source language

`flags` a byte describing the level of detail

`unused`

`asdversion` a byte version number of the debugging data

15.4.1 Lang byte

The language byte codes are defined in the following table:

Language	Code	Description
LANG_NONE	0	Low-level debugging data only
LANG_C	1	C source level debugging data
LANG_PASCAL	2	Pascal source level debugging data
LANG_FORTRAN	3	Fortran-77 source level debugging data
LANG_ASM	4	ARM assembler line number data

Table 15-1: Language byte codes

All other codes are reserved by ARM.

15.4.2 Flags byte

The `flags` byte uses the following mask values:

- debugging data contains line-number information.
- debugging data contains information about top-level variables.
- both of the above

15.4.3 asdversion byte

The `asdversion` byte should be set to 2, the version of this definition.

The flag bytes are followed by the following word-sized fields:

`codestart` Address of first instruction in this section, relocated by the linker.

`datastart` Address of start of static data for this section, relocated by the linker.

`codesize` Byte size of executable code in this section.

`datasize` Byte size of the static data in this section.

`fileinfo` Offset in the debugging area of the fileinfo item for this section (0 if no fileinfo item present). The `fileinfo` field is 0 if no source file information is present.

`debugsize` Total byte length of debug data for this section.

`name or nsyms` String or integer. The `name` field contains the program name for Pascal and Fortran programs. For C programs it contains a name derived by the compiler from the root filename (notionally a module name). In each case, the name is similar to a variable name in the source language. For a low-level debugging section (language = 0), the field is treated as a four-byte integer giving the number of symbols following.

For linker-generated debug data, the fields have these values:

`language`	0
`codestart`	Image$$RO$$Base
`datastart`	Image$$RW$$Base
`codesize`	Image$$RO$$Limit - Image$$RO$$Base
`datasize`	Image$$RW$$Limit - Image$$RW$$Base
`fileinfo`	0
`nsyms`	Number of symbols in the next debugging data.
`debugsize`	Total size of the low-level debugging data including the size of this section item.

Also, the section item is followed by `nsyms symbol` items, each consisting of two words:

`sym` Flags + byte offset in string table of symbol name. `sym` encodes an index into the string table in the 24 least significant bits, and the following flag values in the eight most significant bits, as in ***Table 15-2: Linker-generated debugging data***.

`value` the symbol's value.

Symbol	Offset	Description
ASD_ABSSYM	0	if the symbol is absolute
ASD_GLOBSYM	0x01000000L	if the symbol is global
ASD_TEXTSYM	0x02000000L	if the symbol names code
ASD_DATASYM	0x04000000L	if the symbol names data
ASD_ZINITSYM	0x06000000L	if the symbol names 0-initialized data
ASD_16BITSYM	0x10000000L	bit set if the symbol is a Thumb symbol

Table 15-2: Linker-generated debugging data

Note *The linker reduces all symbol values to absolute values, so that the flag values record the history, or origin, of the symbol in the image.*

Immediately after the symbol table is the string table, in standard AOF format. It consists of:

- a length word
- the strings themselves, each terminated by a NUL (0)

The length word includes the size of the length word, so no offset into the string table is less than four. The end of the string table is padded with NULs to the next word boundary (so the length is a multiple of 4).

15.5 Procedure Items

A procedure item appears once for each procedure or function definition in the source program. Any definitions within the procedure have their related debugging data items between the procedure item and its matching endproc item. After its code and length field, a procedure item contains the following word-sized fields:

`type`	the return type if this is a function, else 0 (see ***15.3 Representation of Data Types*** on page 15-4)
`args`	the number of arguments
`sourcepos`	the source position of the procedure's start (see ***15.3.1 Representation of source file positions*** on page 15-5)
`startaddr`	address of the first instruction of the procedure prologue The `startaddr` field addresses the start of the prologue; it is the instruction where control arrives when the procedure is called.
`entry`	address of the first instruction of the procedure body The `entry` field addresses the first instruction following the procedure prologue; it is the first address where a high-level breakpoint could sensibly be set.
`endproc`	offset of the related endproc item
`fileentry`	offset of the file list entry for the source file
`name`	string

15.5.1 Type items

A type item is used to describe a named type in the source language (eg. a typedef in C). After its code and length field, a type item contains two word-sized fields:

`type`	a type word (described in ***15.3 Representation of Data Types*** on page 15-4)
`name`	string

15.5.2 Endproc items

An endproc item marks the end of the debugging data items belonging to a particular procedure. It also contains information relating to the procedure's return. After its code and length field, an endproc item contains the following word-sized fields:

`sourcepos`	position in the source file of the procedure's end (see ***15.3.1 Representation of source file positions*** on page 15-5)
`endpoint`	address of the code byte *after* the compiled code for the procedure
`fileentry`	offset of the file-list entry for the procedure's end
`nreturns`	number of procedure return points (may be 0)
`retaddrs`	array of addresses of procedure return code

If the procedure body is an infinite loop, there will be no return point, so `nreturns` will be 0. Otherwise each member of `retaddrs` should point to a suitable location at which a breakpoint may be set *at the exit of the procedure*. When execution reaches this point, the current stack frame should still be for this procedure.

15.5.3 Label items

A label in a source program is represented by a special procedure item with no matching endproc (the endproc field is 0 to denote this). Pascal and Fortran numerical labels are converted by their respective compilers into strings prefixed by $n. For Fortran77, multiple entry points to the same procedure each give rise to a separate procedure item, all of which have the same `endproc` offset referring to the unique, matching endproc item.

15.5.4 Variable items

A variable item contains debugging data relating to a source program variable, or a formal argument to a procedure (the first variable items in a procedure always describe its arguments). After its code and length field, a variable item contains the following word-sized fields:

`type`	type of this variable (see ***15.3 Representation of Data Types*** on page 15-4)
`sourcepos`	the source position of the variable (see ***15.3.1 Representation of source file positions*** on page 15-5)
`storageclass`	a word encoding the variable's storage class
`location`	see explanation below
`name`	string

The following codes define the storage classes of variables:

1 external variables (or Fortran common)
2 static variables private to one section
3 automatic variables
4 register variables
5 Pascal 'var' arguments
6 Fortran arguments
7 Fortran character arguments

The meaning of the location field of a variable item depends on the storage class:

- an absolute address for static and external variables (relocated by the linker)
- a stack offset (offset from the frame pointer) for automatic and var-type args
- an offset into the argument list for Fortran argument
- a register number for register variables (the eight floating-point registers are 16..23)

The `sourcepos` field is used by the debugger to distinguish between different definitions that have the same name (eg. identically named variables in disjoint source-level naming scopes such as nested blocks in C).

15.5.5 Struct, union, and class items

A struct item is used to describe a structured data type (eg. a struct in C or a record in Pascal). A class item is used to describe a C++ class type. A union item is used to describe a union type. All have the same format. Note that the C or C++ tag for a struct, union, or class is not represented in the debug table.

After its code and length field, a struct item contains the following word-sized fields:

`fields`	the number of fields in the structure
`size`	total byte size of the structure
`fieldtable...`	an array of `fields` struct field items

Each struct field item has the following word-sized fields:

`offset`	byte offset of this field within the structure
`type`	a type word (described in ***15.3 Representation of Data Types*** on page 15-4)
`name`	string

Earlier versions of the ARM tools described union types by struct items in which all fields have 0 offsets.

For C and C++ bit fields, the type part of the type word identifies a bitfield item.

15.5.6 Array items

An array item is used to describe a one-dimensional array. Multi-dimensional arrays are described as *arrays of arrays*. Which dimension comes first is dependent on the source language (which is different for C and Fortran). After its code and length field, an array item contains the following word-sized fields:

`size`	total byte size of the array. If the size field is zero, debugger operations which affect the whole array, rather than individual elements of it, are forbidden.
`flags`	(see below)
`basetype`	a type word (described in ***15.3 Representation of Data Types*** on page 15-4)
`lowerbound`	constant value or location of variable
`upperbound`	constant value or location of variable

The following mask values are defined for the `flags` field:

`ARRAY_UNDEF_LBOUND`	1	lower bound is undefined
`ARRAY_CONST_LBOUND`	2	lower bound is a constant
`ARRAY_UNDEF_UBOUND`	4	upper bound is undefined
`ARRAY_CONST_UBOUND`	8	upper bound is a constant
`ARRAY_VAR_LBOUND`	16	lower bound is a variable
`ARRAY_VAR_UBOUND`	32	upper bound is a variable

15.5.7 Bounds

Note *A variable item may be used to describe a location known to the compiler, which need not correspond to a source language variable.*

`undefined`	No information about it is available.
`constant`	Its value is known at compile time. In this case, the corresponding bound field gives its value.
`variable`	The offset field identifies a variable debug item describing the location containing the bound. In a debug area in an object file, the offset field contains the offset from the start of the debug area to the variable item.

15.5.8 Subrange items

A subrange item is used to describe a subrange typed in Pascal. It also serves to describe enumerated types in C, and scalars in Pascal (in which case the base type is understood to be an unsigned integer of appropriate size). After its code and length field, a subrange item contains the following word-sized fields:

`lb` low bound of subrange

`hb` high bound of subrange

`sizeandtype` encodes the byte size of container for the subrange (1, 2 or 4) in its least significant 16 bits, and a simple type code (see ***15.3 Representation of Data Types*** on page 15-4) in its most significant 16 bits. The type code refers to the base type of the subrange; for example, a subrange 256..511 of unsigned short may be held in one byte.

15.5.9 Set items

A set item is used to describe a Pascal set type. Currently, the description is only partial. After its code and length field, a set item consists of a single word:

`size` byte size of the object

15.5.10 Enumeration items

An enumeration item describes a Pascal or C enumerated type. After its code and length word, the description of a *contiguous enumeration* contains the following word-sized fields:

`type` a type word describing the type of the container for the enumeration (see ***15.3 Representation of Data Types*** on page 15-4)

`count` the cardinality of the enumeration

`base` the first (lowest) value (may be negative)

`nametable` a character array containing *`count`* name strings

The description of a discontiguous enumeration (such as the C enumeration enum bits {bit0=1, bit1=2, bit2=4, bit3=8, bit4=16}) contains the following fields after its code and length word:

`type` as above

`count` as above

`nametable` a table of *`count`* (value, name) pairs

Each nametable entry has the following format (which is variable in length):

`val` the enumerated value (1/2/4/8/16 in the example)

`name` string (may be several words long)

15.5.11 Function declaration items

After its code and length word, a function declaration item contains the following fields:

- `type` — a type word (see ***15.3 Representation of Data Types*** on page 15-4) describing the return type of the function or procedure
- `argcount` — the number of arguments to the function
- `args` — a sequence of *`argcount`* argument description items

Each argument description item contains the following:

- `type` — a type word (see ***15.3 Representation of Data Types*** on page 15-4) describing the type of the argument
- `name` — string (may be several words)

An argument descriptor need not be named; in this case the length of the name is zero, and the name field is a single zero word.

15.5.12 Begin and end naming scope items

These debug items are used to mark the beginning and end of a naming scope. They must be properly nested in the debug area. In each case, after the code and length word, there is one word-sized field:

- `codeaddress` — address of the start/end of scope (which is determined by the code word)

15.5.13 Bitfield item

A bitfield item describes a bitfield member of a C or C++ struct, union, or class.
After the code and length field, a bitfield item contains the following fields, which are then followed by two zero bytes to pad to a word boundary:

- `type` — a type word describing the type of the field
- `container` — a type word describing the type of the field's container
- `size` — a byte containing the size of the field in bits
- `offset` — a byte containing the offset from bit 0 of the containing value of bit 0 of the field

15.5.14 Macro definition item

A macro definition item describes a C or C++ preprocessor macro definition (#define).
After the code and length field, a macro definition item contains the following fields:

- `fileentry` — offset of the file list entry for the source file containing the macro definition
- `sourcepos` — the source position of the macro definition
- `body` — the offset of the replacement for the macro (not a string, but the characters of the replacement, terminated by a zero byte)

`argcount`	a word containing the number of arguments for the macro. For object-type macros, this field has the value -1.
`argtable`	offset of a description of the macro's argument names. This contains `argcount` strings. The field is zero if the macro has no arguments.
`name`	string

Note *The* `body` *and* `argtable` *offsets are contained within the macro definition item, but the offset is an offset within the containing section.*

15.5.15 Macro undefinition item

A macro undefinition item describes a C or C++ preprocessor macro undefinition (#undef). After the code and length field, a macro undefinition item contains the following fields:

`fileentry`	offset of the file list entry for the source file containing the macro undefinition
`sourcepos`	the source position of the macro undefinition
`name`	string

15.5.16 Fileinfo items

A fileinfo item appears once per section, after all other debugging data items. If the fileinfo item is too large for its length to be encoded in 16 bits, its length field must be written as 0 (since this is the last item in a section and the section header contains the length of the whole section, the length field is strictly redundant).

Each source file is described by a sequence of *fragments*. Each *fragment* describes a contiguous region of the file, within which the addresses of compiled code increase monotonically with source file position. The order in which fragments appear in the sequence is not necessarily related to the source file positions to which they refer.

Note *For compilations which make no use of the #include facility, the list of fragments may have only one entry, and all line-number information can be contiguous.*

After its code and length word, the fileinfo item is a sequence of file entry items of this format:

`len`	length of this entry in bytes (including the length of the following fragments)
`date`	date and time when the file was last modified (may be 0, indicating not available, or unused). If present, the `date` field contains the number of seconds since the beginning of 1970 (the UNIX date origin).
`filename`	string (or "" if the name is not known)
`fragment data`	see below

Following the final file entry item, a single 0 word marks the end of the sequence.

The fragment data is a word giving the number of following fragments, followed by a sequence of fragment items:

n	number of fragments following
fragments...	n fragment items

Each fragment item consists of five words, followed by a sequence of byte pairs and halfword pairs, formatted as follows:

size	length of this fragment in bytes (including length of following lineinfo items)
firstline	linenumber
lastline	linenumber
codestart	pointer to the start of the fragment's executable code
codesize	byte size of the code in the fragment
lineinfo...	a variable number of bytes matching line numbers to code addresses

Each `lineinfo` item describes a source statement and consists of a pair of (unsigned) bytes, possibly followed by two or three (unsigned) halfwords (each halfword has the byte ordering appropriate to the target memory system's endianness or byte sex).

The short form (pair of bytes) lineinfo item is as follows:

codeinc	# bytes of code generated by this statement. If `codeinc` is greater than 255, or `lineinc` is required to describe a line number change greater than 63 or a column change greater than 191, then both bytes are written to describe 0 increments, and the real values are given in the following two or three (unsigned) halfwords.
lineinc	# source space occupied by this statement. `lineinc` describes how to calculate the source position (line, column) of the next statement from the source position of this one. If `lineinc` is in the range 0 <= `lineinc` < 64, the new position is (line+`lineinc`,1). If `lineinc` >= 64, the new position is (line,column+`lineinc`-64).

The number of bytes of code generated for a statement may be zero, provided that the line increment is non-zero (such an item may describe a block end or block start).

It is not possible to describe a statement which generates no code and no line number increment, as encoding is used as an escape to the long form lineinfo items.

Note *There are two ways to describe zero increments: zero lines and zero columns, which serves to discriminate between the two halfword and three halfword forms.*

If the starting column for the next statement is 1, the two-halfword form is used, which in effect is a triple of halfwords as follows:

zero	2 zero bytes
lineinc	# source lines occupied by this statement
codeinc	# bytes of code generated by this statement

Note *The order of the* `lineinc` *and* `codeinc` *halfwords is the reverse of the corresponding bytes.*

If the starting column for the next statement is not 1, the three-halfword form is used, which in effect is a quadruple of halfwords, as follows:

```
codeinc = 0, lineinc = 64
lineinc          # source lines occupied by this statement
codeinc          # bytes of code generated by this statement
newcol           starting column for the next statement
```

Note *Again, the order of the* `lineinc` *and* `codeinc` *halfwords is the reverse of the corresponding bytes. In addition, the column item here is the absolute column number for the next statement, and not an increment as in the two byte form.*
(This encoding of `lineinfo` *items is an incompatible change from the previous format (version 2): in that format,* `lineinc` *in a two byte lineinfo item always describes a line increment, and accordingly, there is no four halfword form. Programs interpreting asd tables should interpret lineinfo items differently according to the table format in the section item.)*

15.5.17 Map fragment items

An FP map fragment item describes the offsets from the stack pointer of the “virtual frame pointer” in functions without a frame pointer. The stack offsets in variable items are offsets from its virtual frame pointer. Functions may have no frame pointer because a no-fp APCS variant has been selected, or because they are sufficiently simple leaf functions with an fp APCS variant. In the latter case, FP map fragments will be generated only for those functions which actually use the stack.

FP map fragments are generated regardless of whether other debug tables are being generated, in a separate debug area (whether or not the `-g` complier option is used).

After the code and length field, an FP map fragment item contains the following fields:

bytes — exact size of the item in bytes (the length part of the code and length field is rounded up to a word boundary)

codestart — address of the first word of code described

saveaddr — address of the instruction saving callee-save registers.
(0 if there is none within the area of code described)

codesize — size of the area of code described

initoffset — offset of the virtual FP from the SP at the start of the code area described.
These items are followed by a variable number of FPInfo fields, which take one of these forms:

`byte codeinc`	size of code area described by this item (unsigned)
`byte offsetinc`	change in the virtual FP offset for the next item (signed)

or, two zero bytes:

`short codeinc`	(unsigned)
`short offsetinc`	(signed)

16 ELF File Format

This section describes the ELF file format.

ELF File Format

16.1 Overview of ELF File Format

It is assumed that the reader is familiar with ELF version 1. This section only describes options taken by ARM in its executable file format; unless otherwise stated, Executable ARM ELF files are as defined in the TIS Portable Formats Specification, Version 1.1.

16.1.1 Object file format

ELF describes three types of Object File:

- relocatable file
- executable file
- shared object file

In general, an ELF Object File has the following organization:

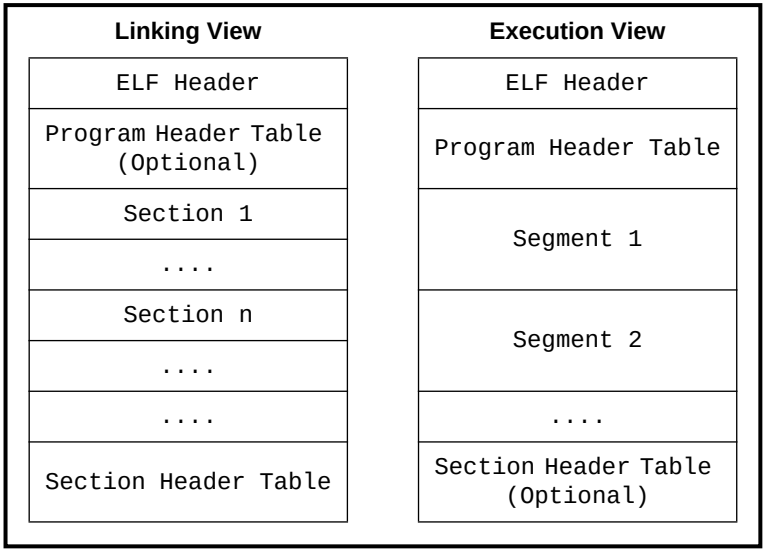


Figure 16-1: ELF object file organization

Section header table

The view of an Object File as a series of named Sections is used by a linker or debugger. Several Sections are denoted “special” and have names reserved which define their contents. For example:

- .bss
- .text

The Section Header Table gives access to such sections.

Program header table

The view of an Object File as a series of Segments is typically used by a loader in order to create an executable process image for a particular runtime environment.

The Program Header Table gives access to such Segments.

16.1.2 Executable ARM ELF File Layout

The ARM linker is used to produce an Executable ARM ELF file. For simple cases, it lays out the file as shown in ***16.2 Generic ELF File Layout*** on page 16-4.

Extra information is encoded in the file for scatter-loaded and overlayed executables and these special cases are described in ***16.3 Scatter-loaded Executables*** on page 16-8.

Only Segments will form part of the final executable image. Sections are included in the Executable to provide further information on how to create the executable image.

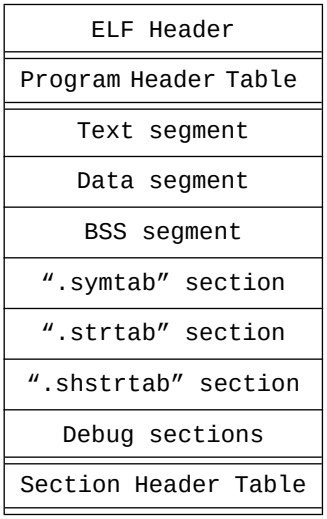
16.2 Generic ELF File Layout

A simple Executable ARM ELF file has the conceptual layout shown in the diagram on the right.

Note that the actual ordering of the file may be different from that shown, since only an ELF header has a fixed position in the file.

All other parts of the file have a position defined by:

- the ELF header
- the Program Header Table
- the Section Header Table



16.2.1 ARM-specific ELF Header Values

This section describes the values in the ELF header which need to be defined for the ARM target environment. All other values are as specified in the *Tool Interface Standard Portable Formats Specification*:

`e_machine`	is set to `EM_ARM` (defined as 40)	
`e_ident[EI_CLASS]`	is set to `ELFCLASS32`	
`e_ident[EI_DATA]`	is set to:	
	`ELFDATA2LSB`	for little-endian targets
	`ELFDATA2MSB`	for big-endian targets

Note: *The endianness of the target is determined by the endianness of the ELF Object Files submitted to the ARM linker. The linker will produce an error message if presented with object files of mixed endianness.*

16.2.2 Segments

There are three types of Segment:

- Text
- Data
- BSS

Entries for these appear in the Program Header Table.

In a simple Executable ARM ELF file, there is just one of each type of Segment; more complex cases are described in ***16.3 Scatter-loaded Executables*** on page 16-8.

Attributes of these Segments are described below:

Text Segment

Contains the code for the executable.

```
p_type    - set to PT_LOAD
p_vaddr   - load address of the segment
p_paddr   - 0
p_filesz  - size of text segment
p_memsz   - same as p_filesz
p_flags   - PF_X + PF_R
p_align   - 4
```

Data Segment

Contains initialized read-write data for the executable.

```
p_type    - set to PT_LOAD
p_vaddr   - load address of data segment
p_paddr   - 0
p_filesz  - size of data segment
p_memsz   - same as p_filesz
p_flags   - PF_R + PF_W
p_align   - 4
```

BSS Segment

Contains uninitialized data, which should be zeroed either when an image is created, or at program startup by the runtime environment. Note that a BSS Segment is distinguished by having a p_filesz of 0 to indicate that it occupies no space in the executable file.

```
p_type    - set to PT_LOAD
p_vaddr   - load address of BSS data segment
p_paddr   - 0
p_filesz  - 0  (note: occupies no file space)
p_memsz   - size of BSS segment
p_flags   - PF_R + PF_W
p_align   - 4
```

16.2.3 Sections

Under the ELF specification, an executable object file can include a Section Header Table which describes Sections in the file. In Executable ARM ELF, all Executables have at least two Sections, unless the linker has been invoked with `-nodebug`; then the Executable has no Symbol Table or String Table Sections:

- the Symbol Table Section
- the String Table Section

Further Sections may appear in the file, and these are described later in ***16.3 Scatter-loaded Executables*** on page 16-8.

When an Executable contains source-level debugging information, it also includes several Debugging Sections, as described below.

If required, an Executable can be stripped of its Sections, leaving just the Text, Data and BSS Segments. The Section Header Table is also removed.

Symbol Table Section

The Symbol Table Section has the following attributes:

```
sh_name:   ".symtab"
sh_type:   SHT_SYMTAB
sh_addr:   0  (to indicate it is not part of the image)
```

Note: *In Executable ARM ELF we do not set the SHF_ALLOC bit in the sh_flags field, thus indicating that there is no space allocated for the symbol table in the image which will be created from this Executable.*

This symbol table can be used for low-level debugging symbol information.

String Table Section

The String Table Section holds all strings referenced by other Sections in the Executable. In particular it will hold the textual names of entries in the Symbol Table Section. It has the following attributes:

```
sh_name:   ".strtab"
sh_type:   SHT_STRTAB
sh_addr:   0 (to indicate it is not part of the image)
```

Section Name String Table

The Section Name String Table holds the textual names of all sections. It has the following attributes:

```
sh_name:   ".shstrtab"
sh_type:   SHT_STRTAB
sh_addr:   0 (to indicate it is not part of the image)
```

Debugging Sections

ARM Executable ELF supports three types of debugging information held in debugging Sections. A consumer of an ELF executable can distinguish between these three types of debugging information by examining the Section Table for the executable:

- ASD debugging tables

  These provide backwards compatibility with ARM's Symbolic Debugger. ASD debugging information is stored in a single Section in the executable named `.asd`.
- DWARF version 1.0

When DWARF 1.0 debugging information is included by the linker in the ELF executable, the file contains the following ELF Sections, each of which has a Section Header Table entry:

Section Name	Contents
`.debug`	debugging entries
`.line`	`fileinfo` entries
`.debug_pubnames`	table for accelerated access to debug items
`.debug_aranges`	address ranges for compilation units

- DWARF version 2.0

  When DWARF 2.0 debugging information is included by the linker in the ELF executable, the file contains the following ELF sections, each of which has a Section Header Table entry:

Section Name	Contents
`.debug_info`	debugging entries
`.debug_line`	`fileinfo` statement program
`.debug_pubnames`	table for accelerated access to debug items
`.debug_aranges`	address ranges for compilation units
`.debug_macinfo`	macro information (#define / #undef)
`.debug_frame`	call frame information
`.debug_abbrev`	abbreviation table
`.debug_str`	debug string table

Each of the `.debug_*` sections will have type SHT_PROGBITS. These are only included when source-level debugging information is available.

Each entry will have a `sh_addr` member of 0 indicating that the file contains the debugging information, but that this information will not be included in an image created from the executable.

Note: *This means debugging information (albeit maybe only low-level debug symbols) can be kept for a program image residing in ROM without that information appearing in the ROM itself.*

16.3 Scatter-loaded Executables

When scatter loading is used, the ARM linker generates Section Header Table entries for a load region, where the Section name is taken from the load region name as defined in the scatter description.

If the load region contains a mixture of code, data and uninitialized data, there will be more than one Segment generated for that load region:

- each Segment generated will have its own Section Header Table entry, so more than one Section may have the same name
- each Section Header Table entry will have its sh_offset field set to the same value as the p_offset field of the Program Header Table entry for its corresponding Segment.

Each Section Header Table entry for a load region will have the following attributes:

`sh_name:`	name of load region (as given in scatter description)
`sh_type:`	SHT_PROGBITS or SHT_NOBITS (for zero init areas)
`sh_addr:`	same as p_vaddr of corresponding Segment
`sh_offset:`	same as p_offset of corresponding Segment
`sh_flags:`	bit SHF_LOADREGION set

The `p_vaddr` field of each Segment of a scatter-loaded Executable is the load address of the Segment, which need not necessarily be its execution address. Startup code can move (part of) a Segment to its execution address using the symbols:

```
Load$$reg$$Base
Image$$reg$$Base
Image$$reg$$Length
```

as described in the *Software Development Toolkit User Guide (ARM DUI 0040)*.

The Section Header Table entries can be used to generate a separate plain binary file for each load region, using the Section name as the name of the generated file. A tool which does this would need to merge any Sections of the same name, sorting them by address (`sh_addr`). Alternatively a single image can be made simply by using the Segment information held in the Program Header Table.

Note: *Debugging information (if included) for the Executable will refer to memory locations in execution regions rather than load regions.*

17 Other File Formats

This section describes other file formats used by the ARM Software Development Toolkit.

17.1 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, and 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 via an input library)
- an absolute base address (given by the `-Base` option to armlink). This is set to zero if it is not specified.
- complete performance of all relocation directives

Input areas having the read-only attribute are placed at the low-address end of the image; initialized writable areas follow; zero-initialized areas are consolidated at the end of the file where a block of zeros of the appropriate size is written, unless the `-nozeropad` linker option is specified. The application then needs to create the zero-initialized area at runtime.

Note *If the binary file is part of a scatter-loaded application, the zero-initialized areas are not present. The initialization information generated by the linker enables the application initialization to generate the zero-initialized areas at runtime.*

17.2 Extended Intellec Hex Format (IHF)

This format is for small (< 64Kb) images, such as those destined for ROM. IHF is essentially a plain binary format, encoded as 32-bit hex values and checksummed. All the restrictions of plain binary format apply to the generation of IHF.

The following linker option supports IHF:

`-ihf` Generates a plain binary image encoded in VLSI Extended Intellec Hex Format. The output is ASCII-coded, is always big-endian and is suitable for driving the Compass integrated circuit design tools (see ***Chapter 17, Other File Formats*** for details).

The linker is described fully in ***Chapter 3, Linker***.

Index

Index

Symbols

Numerics

A

B

C

D

E

F

H

I

L

M

N

O

P

R

S

T

U

W

Z