ARM Software Development Toolkit Version 2.0

Reference Manual

Document Number: ARM DUI 0020D
Issued: June 1995

EUROPE
Advanced RISC Machines Limited
Fulbourn Road
Cherry Hinton
Cambridge CB1 4JN
Telephone: +44 1223 400400
Facsimile: +44 1223 400410
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
Suite 5, 985 University Avenue
Los Gatos
California 95030
Telephone: +1 408 399 5199
Facsimile: +1 408 399 8854
Email: info@arm.com

Change Log

Issue	Date	By	Change
Prelim1	Nov 94	BJH/EH	Created
B01	Jan 95	AW	Changes to reflect review comments; change of title
B00	Feb 95	AW	B01 comments incorporated
C	April 95	PB	Updated, profiler chapter added
D	June 95	BJH/PO	Edited

Contents

Contents

Contents

Contents

Contents

1 Introduction

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

Introduction

1.1 About This Manual

1.1.1 Overview

This manual deals with the following topics:

- the components of the ARM Software Development Toolkit
- an overview of each of the ARM Software Tools and how to invoke them
- an introduction to the Runtime Libraries
- ARM Assembly Language
- Thumb Assembly Language
- simulating a processor
- using the symbolic debugger

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

1.1.2 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-italic`	shows text which must be substituted with user-supplied information: this is most often used in syntax descriptions
Oblique	is used to highlight important notes and ARM-specific terminology

Thumb: Boxes like this contain information that applies specifically to Thumb-aware variants of the ARM toolkit.

1.2 Release Components

1.2.1 Programming and modelling 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: ➲*Chapter 2, C Compiler*.
`tcc`	The Thumb C compiler: ➲*Chapter 2, C Compiler*.
`armasm`	The ARM assembler ➲*Chapter 3, Assembler*.
`armlink`	The ARM linker: ➲*Chapter 6, Linker*.
`armsd`	The ARM symbolic debugger: ➲*Chapter 7, Symbolic Debugger*.
`armlib`	The ARM object-file librarian: ➲*Chapter 9, ARM Librarian*.
`decaof`	The ARM-Thumb object-file decoder/dissasembler: ➲*Chapter 10, ARM Object Format Decoder*.
`topcc`	A PCC to ANSI C dialect conversion tool: ➲*Chapter 11, ANSI to PCC C Translator*.
`reconfig`	The ARM tools reconfiguration utility: ➲*Chapter 12, ARM Tool Reconfiguration Utility*.
`armmake`	The ARM make utility: ➲*Chapter 13, ARM make Utility*.

1.2.2 Retargetable libraries

Two retargetable libraries are supplied:

- The ARM ANSI C library, supplied in both source form and as an object library.
- The minimal standalone C runtime support system. This is supplied as ARM Assembler, and is all that is needed to support unhosted, ARM-targeted C.

For further information see ➲*15.1 An Introduction to the Run-Time Libraries* on page 15-2. Details about porting the ARM-targeted ANSI C Library is given in ➲*15.2 Porting the ARM C Library* on page 15-3.

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

1.3 Feedback

1.3.1 Feedback on the Software Development Toolkit

If you have feedback on the Software Development Toolkit, please contact either your supplier or ARM Ltd. You can send feedback via e-mail to: xdevt@armltd.co.uk.

In order to help us give a rapid and useful response, please give:

- details of which hosting and release of the ARM software tools you are using
- a small sample code fragment which reproduces the problem
- a clear explanation of what you expected to happen, and what actually happened

1.3.2 Feedback on this manual

If you have feedback on this manual, please send it via e-mail to: documentation@armltd.co.uk, giving:

- the manual's revision number
- the page number(s) to which your comments refer
- a concise explanation of the problem

General suggestions for additions and improvements are also welcome.

2 C Compiler

This section describes the ARM and Thumb C compilers.

C Compiler

2.1 Introduction

This chapter includes all the information you need to make effective use of the ARM C system. It is not intended to be an introduction to C and does not try to teach programming in C, nor is it a reference manual for the C standard.

The ARM instruction set is documented separately in ARM datasheets. Refer to the datasheet for the ARM variant you are using.

For details of how to use the ARM assembler, please refer to *Chapter 3, Assembler*.
ARM assembly language is documented in *Assembly Language Overview* on page 3-6. If you only need to understand the assembly language output by the C compilers, refer to the datasheet for your device.

2.1.1 Recommended texts

C programming guides

Because the ARM C compiler is a compiler for ANSI C, these books are especially relevant but, where noted in the list, the second edition must be obtained for coverage 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.
- Kernighan, B.W. and Ritchie, D.M.,*The C Programming Language* (second edition, 1988). Prentice-Hall, Englewood Cliffs, NJ, USA. ISBN 0-13-110362-8.
  Kernighan and Ritchie is the original C bible, updated to cover the essentials of 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 also available from ANSI as X3J11/90-013.

  The standard is available from the national standards body (eg. AFNOR in France, ANSI in the USA).

2.2 About the ARM C Compiler

The ARM C compiler is a mature, industrial-strength compiler, based on Codemist Limited's multi-target, multi-language compiler suite (also known as the NorCroft C compiler). Some derived compilers are used by, or distributed by:

- Advanced RISC Machines (for the ARM processor)
- Acorn Computers (for their ARM-based personal workstations)
- INMOS (for the Transputer)
- Hitachi (used internally on IBM 370 compatibles)
- Perihelion Software (for their Helios portable operating system)

2.2.1 Compiler variants

There are two variants of the ARM C compiler:

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

Since 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, please refer to ➲ARM/Thumb interworking* on page 2-57.

2.2.2 Source language modes

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

pcc mode

You select pcc mode from the compiler's command line. pcc mode accepts the dialect of C used by Berkeley Unix. In this mode, the compiler has been used to build a complete ARM-based BSD Unix system (the RISCiX system, marketed by Acorn Computers Limited, which has also achieved X/Open branding).

ANSI mode

In ANSI mode, the ARM C compiler has been tested against release 5.00 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 fails in only two trivial ways.

Both are failures to produce required diagnostics:

- an empty initialiser for an aggregate of complete type is not diagnosed
  ```
  int x[3] = {};
  ```
- signed integer constant overflow is not diagnosed, but merely warned of
  ```
  case INT_MAX+1: ...
  ```

Wherever possible, the ARM C compiler adopts widely-used command-line options which should be familiar to users of both Unix and DOS.

2.3 Invoking the Compiler

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

```
toolname options sourcefile
```

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

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

2.3.1 Command-line options

Many aspects of the compiler's operation can be controlled via command-line options. All options are prefixed by a minus sign.

There are two classes of option:

Keywords	are recognised in upper case or lower case
Flags	A flag is a single letter. Its case is sometimes important to the ARM C compiler.

Case-sensitive options

Because some systems (such as Unix) are case-sensitive, the case of flags is most important when you are building portable makefiles. By using the conventions common to many C compilers, you can move a makefile between different environments at minimum cost.

2.4 Using the ARM C Compiler

This section introduces some key concepts.

2.4.1 File naming conventions

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

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

For example, `something.c` names the C source of `something`.

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

Host systems without filename extensions

Some host systems have no filename extensions and no extension convention. On such systems, files may be stored in folders (sub-directories) named `c`, `h`, `o` and `s`. However, the compiler still understands the `something.c` notation, both on its command line and when processing the names of `#include` files, and it translates names written in standard form to host system filenames.

For example, under Acorn's RISC OS system the source `something.c` is actually stored in the file called `something` in subdirectory `c`. Note, however, that under RISC OS, listing files are by default created in an `l` directory, and not a `lst` directory, as might be expected.

Portability

Portability is an increasingly important issue in the C world. ARM C system supports the use of multiple file-naming conventions on one host.

In general, follow these guidelines:

- restrict the name of a file or directory to a maximum of 8 lower-case letters and digits, beginning with a letter
- ensure that extensions are no longer than 3 letters and digits long
- make embedded path names relative, rather than absolute

In each environment the ARM C system supports:

- native filenames
- pseudo Unix filenames
- Unix filenames

A pseudo Unix filename has the format:

```
host-volume-name:/rest-of-unix-file-name
```

Determining how to parse a name is done heuristically as follows:

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

This filename interpretation only succeeds if certain rules are adhered to by program authors. For example, under DOS, a name may not exceed 8 characters in length and character case is not significant.

2.4.2 Filename validity

The compiler does not check whether the filenames given are acceptable to the host's filing system. If the filename is not acceptable, the compiler will report that it could not be opened but will give no further diagnosis.

2.4.3 Object files

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

A C source file (`something.c`) is compiled into an object file (`something.o`) written in ARM Object Format (AOF). AOF is defined in ➲*ARM Object Format* on page 21-10.

2.4.4 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. By placing a filename in angle brackets, you indicate that the included file is a system file and ensure that the compiler looks first in its built-in filing system. In this example, the ARM C compiler *does not* look for system files in the current directory by default:

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

By enclosing a filename in double quotes in its `#include` directive, you indicate that it is not a system file. In this example, the ARM C compiler *does* look for non-system files 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 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 *The current place*, below) (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)

Note: *The in-memory filing system can be specified explicitly by -I or -j by using the 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 old current place. At each instant, there is a stack of current places corresponding to the stack of nested `#include` directives.

For example, let us 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, will be sought relative to `/me/include/sys`.

This is the search rule used by BSD Unix systems. If required, the stacking of current places can be disabled with the compiler -fK option, which makes the compiler use the search rule described originally by Kernighan and Ritchie in *The C Programming Language*. Under this rule all non-rooted user includes are sought relative to the directory containing the source file being compiled.

2.5 Keyword Options

`-help`	Give a summary of the compiler’s command line options.
`-pcc`	Compile (BSD 4.2) portable C compiler C. This dialect is based on the original Kernighan and Ritchie (K&R) 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. For more details see section ➲*PCC Compatibility Mode* on page 2-47.
`-fussy` or `-strict`	Be extra strict about enforcing conformance to ANSI standard or pcc conventions (for example, prohibit the volatile qualifier in pcc mode).
`-list`	Create 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 ➲*2.6.6 Controlling additional compiler features* on page 2-13).
`-via` *`file`*	If this option is specified, the file is opened and more command line arguments are read in from it. This is intended mainly for hostings (such as the PC) where command-line length is severely limited.
`-errors` *`file`*	This writes the compiler error output to file. This is useful in an MS-DOS environment (or any host for which `stderr` cannot be easily redirected) when error output needs to be logged.
`-littleend` or `-li`	Compile code for an ARM operating with little-endian memory (least significant byte has lowest address).
`-bigend` or `-bi`	Compile code for an ARM operating with big-endian memory (most significant byte has lowest address).
	By default, the ARM C compiler compiles code with the same byte order as the host system. However, most releases of the ARM C compiler, for most hosts, allow this default to be configured when the compiler is installed, so it is not usually necessary to use either the `-bigend` or `-littleend` option (see ➲*About reconfig* on page 12-2).
`-apcs [3]q`*`ualifiers`*	Specify which variant of the ARM Procedure Call Standard is to be used by the compiler. The default is set up when the compiler is configured, and for ease of use can be reconfigured using the reconfig tool—see ➲*About reconfig* on page 12-2. Alternatively the default can be changed by use of this keyword option.

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

`/26[bit]`	26-bit APCS variant.*
`/32[bit]`	32-bit APCS variant.*
`/reent[rant]`	Re-entrant APCS variant.*
`/nonreent[rant]`	Non re-entrant APCS variant.*
`/swst[ackcheck]`	Software stack checking APCS variant.*
`/noswst[ackcheck]`	No software stack checking APCS variant.*
`/fp`	Use a dedicated frame-pointer register.*
`/nofp`	Do not use a frame-pointer.*
`/fpe2`	Floating-point emulator 2 compatibility.*
`/fpe3`	Floating-point emulator 3 compatibility.*
`/fpr[egargs]`	FP arguments passed in FP registers.*
`/nofpr[egargs]`	FP arguments are not passed in FP registers.*
`/inter[work]`	Compile code for ARM/Thumb interworking. See ➲*ARM/Thumb interworking* on page 2-57.
`/nointer[work]`	Do not compile code which is suitable for ARM/Thumb interworking.
`/softfp`	Call software floating point library functions.*
`/hardfp`	Generate ARM coprocessor instructions for floating point (may also specify `fpe2`/`fpe3` and `fpr`/`nofpr`.)*

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

2.6 Flag Options

The flag options are listed below. Some of these 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 show the case of the letter that would be accepted by a Unix C compiler.*

The descriptions are divided into the following subsections, so that flags controlling related aspects of the compiler's operation are grouped together:

- Controlling the linker
- Preprocessor flags
- Controlling code generation
- Controlling warning messages
- Suppressing error messages
- Controlling additional compiler features

2.6.1 Controlling the linker

`-c`	Do 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` option.

2.6.2 Preprocessor flags

`-I`*`directory-name`*	adds the specified directory to the list of places which are searched for included files (after the in-memory or source file directory, depending on the type of include file). The directories are searched in the order they are given, by multiple `-I` options. See *Included Files* on page 2-6 for full details.
`-j`*`directory-list`*	is a comma-separated list of search directories. This option adds the list of directories specified to the end of the search path (for example, after all directories specified by `-I` options), but otherwise in the same way as `-I`. It also makes the compiler search the in-memory filing system *after* all other searches have failed. Note that the in-memory filing system can be specified in `-I` and `-j` options by `:mem`. `-j` is an ARM-specific flag and is not portable to other C systems. There may be at most one `-j` option on a command line. See *Included Files* on page 2-6 for full details.

`-E`	If this flag is specified, only the preprocessor phase of the compiler is executed. The output from the preprocessor is sent to the standard output stream. It 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`	When used in conjunction with `-E` above, `-C` retains comments in preprocessor output. Note that this option is different from the `-c` flag, which suppresses the link operation.
`-M`	If this flag is specified, only the preprocessor phase of the compiler is executed (as with armcc `-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 xxx.c >> Makefile` where *`toolname`* is either `armcc` or `tcc`.
`-D`*`symbol=value`*	Define *`symbol`* as a preprocessor macro, as if by a line `#define` *`symbol`* value at the head of the source file.
`-D`*`symbol`*	Define *`symbol`* as a preprocessor macro, as if by a line `#define` *`symbol`* at the head of the source file.
`-U`*`symbol`*	Undefine *`symbol`*, as if by a line `#undef symbol` at the head of the source file.

2.6.3 Controlling code generation

`-g`*`Letters`* This flag specifies that debugging tables for use by the ARM Source Level Debugger (armsd) should be generated. It is followed by an optional set of letters which specify the level of information required. If no letters are present, any available information is generated. However, the tables can occupy large amounts of memory, so it can be useful to limit what is included, as follows.

`-gf`	Generate information on functions and top-level variables (those declared outside of functions) only.
`-gl`	Generate information describing each line in the source file(s).
`-gv`	Generate information describing all variables.

The last three modifiers may be specified in any combination.
For example: `-gfv`.

`-o` *`file`*	The argument to the `-o` flag gives the name of the file which will hold 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`	Perform optimisations to reduce image size at the expense of increased execution time.
`-Otime`	Perform optimisations to reduce execution time at the expense of a larger image.
`-S`	If the `-S` flag is specified, no object code is generated, but a listing of the assembly language is written to a file. By default, the file is called `name.s` in the current directory (where `name.c` is the name of the source file stripped of any leading directory names). The default can be overridden using the `-o` flag.

2.6.4 Controlling warning messages

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.

`-W` If no modifier letters are given, all warnings are suppressed. If one or more letters follow the flag, then only the warnings controlled by those letters are suppressed.

`-Wa` Give no “Use of = in a condition context” warning. This warning is given when the compiler encounters a statement such as:

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

where it is possible that the author really did intend

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

or that the author intended the following, but missed a key stroke.

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

In new code, avoid the deliberate use of `if (a = b) ...`

This warning is also suppressed in `-pcc` mode.

`-Wd` Give no “Deprecated declaration foo() - give arg types” message, given when a declaration without argument types is encountered in ANSI mode (the warning is suppressed in `-pcc` mode).

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` Give no “Inventing extern int foo()” message, which may be useful when compiling old-style C in ANSI mode. Warning is suppressed in `-pcc` mode.

`-Wn`	Give no “Implicit narrowing cast” warning. 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 C system in which ints occupy 16 bits and longs 32 bits (as is common on the IBM PC, Apple Macintosh, etc.)
`-Wp`	Give no “non-ANSI #include <...>” warning. ANSI requires that you should 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.
`-Wv`	Give no “Implicit return in non-void context” warning. 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. Because the practice is widespread in old-style C, the warning is suppressed in `-pcc` mode.

2.6.5 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).

`-e`*`letters`*	Suppresses a range of compile time errors.
`-ec`	Suppresses all implicit cast errors, eg. “implicit cast of non-0 int to pointer”.
`-ep`	Suppresses the error which occurs if there are extraneous characters at the end of a preprocessor line.
`-ez`	Suppresses the error if a zero-length array is used.
`-ef`	Suppresses errors for unclean casts such as short to pointer.
`-ei`	Suppresses syntax checking for skipped `#if` statements.
`-el`	Suppress errors about linkage disagreements where functions are implicitly declared extern and later defined as static.

2.6.6 Controlling additional compiler features

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

`-f`*`Letters`*	The `-f` flag described in this section controls a variety of compiler features, including certain checks more rigorous than usual. Like the `-W` flag it is followed by a string of modifier letters. At least one letter is required, though several may be given at once, for example, `-ffah`.

`-fa`

Check 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 the following:

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

Here, we know that `value` is read only if `initialised` has been set. As this is a semantic deduction, not a data flow implication, `-fa` will report an anomaly. In general, it is useful to check all code using `-fa` at some stage during its development.

`-fc`

Enable 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).

`-fe`

Check 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 a portability aid.

`-ff`

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

`-fh`

Check 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 in-line in a C source file) then these checks directly support good modular programming practices.

`-fi`	In the listing file, list the lines from any files included with directives of the form: `#include "file"`
`-fj`	As above, but for files included by lines of the form: `#include <file>`
`-fk`	Use K&R search rules for locating included files (the current place is defined by the original source file and is not stacked; see section *Included Files* on page 2-6 for details).
`-fm`	Report on preprocessor symbols defined but not used during the compilation.
`-fn`	Embed function names in the code area (see `-ff` option). This improves the readability of the output produced by the stack backtrace run-time 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 *Controlling code generation* on page 2-11).
`-fp`	Report on explicit casts of integers into pointers, eg. `char *cp = (char *) anInteger;` This warning indicates potential portability problems in future. 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`	List 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 then the unexpanded source text, as written by the user, 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;`.
`-fv`	Report on all unused declarations, including those from standard headers.
`-fw`	Allow string literals to be writeable, 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 re-using a multiple-occurring string literal.

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

`-zp`*`LetterDigit`*
: This flag can be used to emulate `#pragma` directives. The letter and digit which follow it are the same characters that would follow the '-' of a `#pragma` directive. See *Pragma directives* on page 2-50 for details.

`-zr`*`Number`*
: This flag allows the size of most LDMs and all STMs to be controlled between the limits of 3 and 16 registers transferred. This can help control interrupt latency where this is critical.

2.7 Processor Selection Options

`-`*`processor`*
: Tells the compiler to compile 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.

`-`*`processor`* should be specified in the form `-ARM`*`N`*`[`*`options`*`]` where *`N`* is the processor number, for example 6, 60, 600 and *`options`* is a list of single letter processor options, for example T, M.

Currently, the only option that has any effect is the `T` (Thumb) option although future releases of the compiler may recognise other options.

Thumb: If you specify a Thumb-aware processor (eg. `-ARM7TDMI`) to armcc this will not cause `armcc` to generate Thumb code. Instead it will generate ARM code which uses the new halfword ARM instructions.
This option is not available with `tcc` as `tcc` always generates Thumb code.

2.8 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 to programmers.

2.8.1 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 *Standard Implementation Definition* on page 2-24). 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.

Upper and lower case 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.

2.8.2 Data Elements

Type	Size in bits
char	8
short	16
int	32
long	32

Type	Size in bits
float	32
double	64
long double	64 (subject to change)
all pointers	32

Table 2-1: Size of 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.

2.8.3 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`

Table 2-2: ARM C compiler integer ranges

Range	End-point	Hex Representation
LONG_MIN	-2147483648	0x80000000
ULONG_MAX	4294967295	0xffffffff

Table 2-2: ARM C compiler integer 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`

2.8.4 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:

- Structures are aligned on word boundaries.
- 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.
- The only valid types for bitfields are (signed) int and unsigned int. (In `-pcc` mode, char, unsigned char, short, unsigned short, long and unsigned long are also accepted.)
- A bitfield of type int is treated as unsigned by default (signed by default in `-pcc` mode).
- A bitfield must be wholly contained within the 32 bits of an int.
- 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

2.8.5 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).

2.8.6 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.

2.8.7 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.

See *Controlling Floating Point Exceptions from C* on page 16-6.

2.8.8 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.

2.8.9 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 *Implementation limits* on page 2-23. 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	31	256
in external identifiers	6	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
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, below)]
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 2-3: Implementation limits

Note: *When running on 16-bit hosts, the ARM C compiler may impose a limit on the size of an object. 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.*

2.9 Standard Implementation Definition

This section discusses aspects of the ARM C compiler and ANSI C library not defined by the ISO C standard, and which are implementation-defined.

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 the points listed there, under the same headings and in the same order.

2.9.1 Translation

Diagnostic messages produced by the compiler are of the form:

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

where *`severity`* is one of:

`Warning`	This is not a diagnostic in the ANSI sense, but a helpful message from the compiler.
`Error`	This is a violation of the ANSI specification from which the compiler was able to recover by guessing the user's intentions.
`Serious error`	This is a violation of the ANSI specification from which no recovery was possible because the compiler could not reliably guess what was intended.
`Fatal`	This is not really a diagnostic but 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).

2.9.2 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()

The arguments given to `main()` are the words of the command line (not including I/O redirections, covered below), delimited by white space, except where the white space is contained in double quotes. A white space 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 recognised 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, in which case 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 run time in the way shown. For example, if `mycopy` is a program which simply copies the standard input to the standard output, the following line:

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

runs the program, redirecting 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`*	read `stdin` from *`filename`*
`<` *`filename`*	read `stdin` from *`filename`*
`1>` *`filename`*	write `stdout` to *`filename`*
`>` *`filename`*	write `stdout` to *`filename`*
`2>` *`filename`*	write `stderr` to *`filename`*
`2>&1`	write `stderr` to same place as `stdout`
`>&` *`filename`*	write both `stdout` and `stderr` to *`filename`*
`>>` *`filename`*	append `stdout` to *`filename`*
`>>&` *`filename`*	append both `stdout` and `stderr` to *`filename`*

2.9.3 Identifiers

256 characters are significant in identifiers without 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 character $ is also valid in identifiers.

2.9.4 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 these 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 multibyte 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 *Escape codes* on page 2-27.

Escape sequence	Char value	Description
`\a`	`7`	Attention (bell)
`\b`	`8`	Backspace
`\f`	`9`	Form feed
`\n`	`10`	Newline
`\r`	`11`	Carriage return
`\t`	`12`	Tab
`\v`	`13`	Vertical tab
`\xnn`	`0xnn`	ASCII code in hexadecimal
`\nnn`	`0nnn`	ASCII code in octal

Table 2-4: Escape codes

2.9.5 Integers

The representations and sets of values of the integral types are set out in *Data Elements* on page 2-17. 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.

2.9.6 Floating-point types

The representations and ranges of values of the floating-point types have been given in *Data Elements* on page 2-17. Note also that:

- When a floating-point number is converted to a shorter floating point one, it is rounded to the nearest representable number.
- The properties of floating-point arithmetic accord with IEEE 754.

2.9.7 Arrays and pointers

The ISO standard specifies three areas in which the behaviour 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)

2.9.8 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 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

Notes: 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.

2.9.9 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.

2.9.10 Declarators

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

2.9.11 Statements

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

2.9.12 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 2-1: Conventional (non-packed) struct example*.

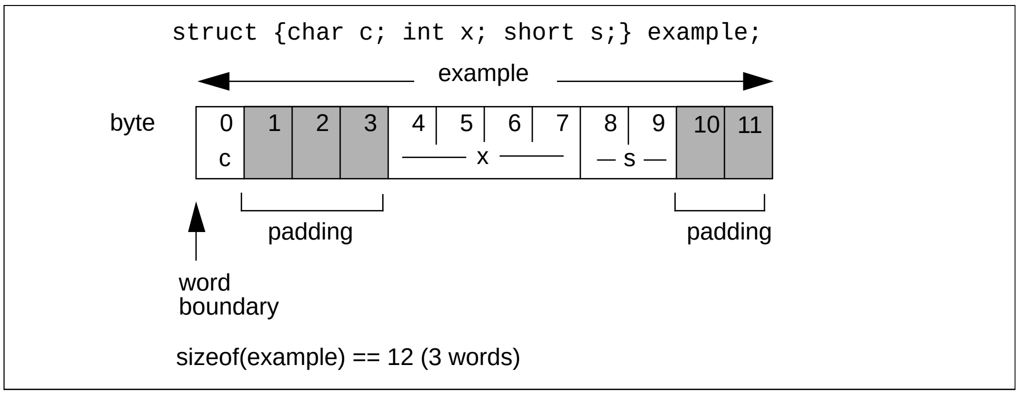

struct {char c; int x; short s;} example;
example
byte
0
1
2
3
4
5
6
7
8
9
10
11
c
x
s
padding
padding
word boundary
sizeof(example) == 12 (3 words)

Figure 2-1: 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 7 instructions and 2 extra work registers). Data accesses via packed structs should be minimised 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 be a new type qualifier: `__packed`.

If you require `packed` rather than `__packed`, then use:

```
#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 non packed 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 inherits the packed attribute, so `__packed` does not have to be specified:

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

In consequence, 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. This does not matter, because the layout in store of top-level objects is not specified by the ANSI standard.

Sub-structs of packed structs

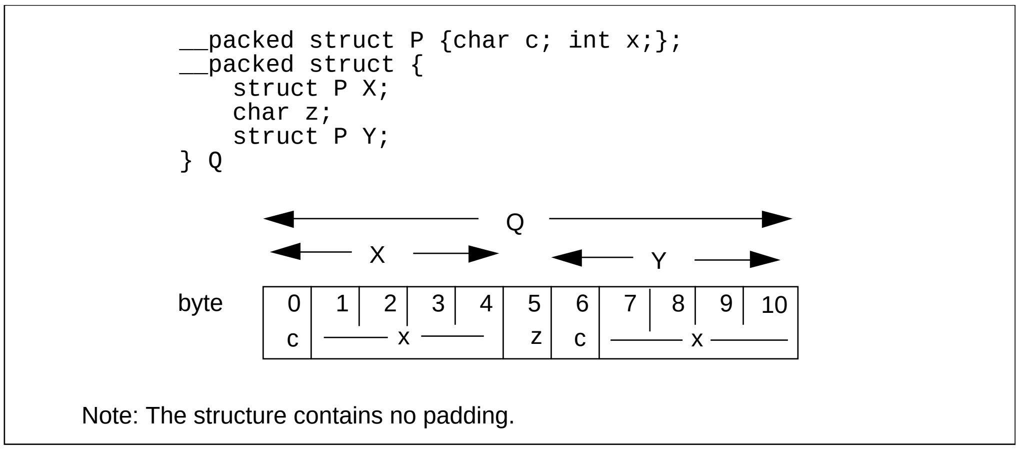
A struct (or union) sub-field of a packed struct or union must be declared to have packed struct (or packed union) type.

```
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 2-2: Sub-struct padding

2.9.13 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.

2.9.14 Enumerations

An object of `enum` type will normally be implemented in the smallest integral type that contains the range of the `enum`.

The type of an `enum` will be `unsigned char`, `signed char`, `unsigned short`, `signed short`, `unsigned int` or `signed int`, according to the range of the `enum`. This feature can reduce the size of the data area.

The command line flag `-fy` sets the underlying type of `enum` to signed int.

2.9.15 Bitfields

The ARM C compiler handles bitfields in the following way:

- a plain bitfield (declared as int) is treated as unsigned int (signed int in pcc mode)
- a bitfield which does not fit into the space remaining in the current int is placed in the next int
- 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

2.9.16 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 *Included Files* on page 2-6. The compiler will accept host filenames or Unix filenames. In the latter case, on non-Unix hosts, the compiler does its best to translate the filename to a local equivalent. See *File naming conventions* on page 2-5 for more details.

The recognized `#pragma` directives and their meanings are described in *Pragma directives* on page 2-50.

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

2.9.17 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 (inclusive):

```
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
192 to 223 (except 215)	upper case
224 to 255 (except 247)	lower case
160 to 191, 215 and 247	punctuation

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 2-5: Mathematical functions

Where `-HUGE_VAL` is written above, 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

The default handling of all recognised signals is to print a diagnostic message and call `exit`. This default behaviour 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 any particular targeting of it may not have):

- 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 (on 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 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.
- `fprintf()` prints %p arguments in hexadecimal format (lower case) 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`

- `calloc()`, `malloc()` and `realloc()`, if the size of area requested is zero, return `NULL`.
- `abort()` closes all open files, and deletes all temporary files.
- 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.

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()`.

2.10 Portability

The C programming language has gained a reputation for being portable across machines, while still providing machine-specific capabilities. However, the fact that a program is written in C gives little indication of the effort required to port it from one machine to another or, indeed, from one C system to another.

The most effort-consuming task is porting between two entirely different hardware environments, running different operating systems with different compilers. Because many users of the ARM C compiler will face this situation, this section deals with the issues that you should be aware of when porting software to or from the ARM C system environment:

- general portability considerations
- the differences between ANSI C and the well-known K&R C
- using the ARM C compiler in pcc compatibility mode
- environmental aspects of portability

In addition, a tool called `topcc` is supplied as part of the ARM Software Development Toolkit. This translates ANSI C to PCC-style C. For details refer to ➲*Chapter 11, ANSI to PCC C Translator*.

If code is to be used on a variety of different systems, there are certain points that you should observe to make porting an easy and relatively error-free process. It is essential to identify and avoid practices which may make software system-specific. The rest of this section documents the general portability issues for C programs.

2.10.1 Fundamental data types

The size of fundamental data types (such as `char`, `int`, `long int`, `short int` and `float`) depend mainly on the underlying architecture of the machine on which the C program is to run. Compiler writers usually implement these types in a way which is natural for the target.
For example, Release 5 of the Microsoft C Compiler for DOS has `int`, `short int` and `long int`, occupying 2, 2 and 4 bytes respectively, while the ARM C Compiler uses 4, 2 and 4 bytes, respectively.

Certain relationships are guaranteed by the ISO C standard; for example:

```
(sizeof(long int) >= sizeof(short int))
```

but code which makes any assumptions about whether `int` and `long int` have the same size, are not portable.

A common non-portable assumption is embedded in the use of hexadecimal constant values:

```
int i = 0xffff;/*-1 if sizeof(int) == 2;
        65535 if sizeof(int) == 4... */
```

Argument passing

In non-ANSI dialects of C there are pitfalls with argument passing. For example:

```
int f(x)
long int x;
{...}
```

and the (careless) invocation of `f()`:

```
f(1);  /*f(1L) was intended/required */
```

If `sizeof(int) == sizeof(long int)` this gives no problem. If not, there may be a catastrophe.

A dual problem afflicts the format string of the `printf()` family, even in ANSI C. For example:

```
long int l1, l2, l3;
...
printf("L1 = %d, L2 = %d, L3 = %d\n", l1, l2, l3);
       /* "...%ld...%ld...%ld..." is intended/required */
```

Again, this causes problems if `sizeof(int) != sizeof(long)`.

Character signs

Another common assumption is about the signedness of characters, especially if chars are expected to be 7-bit rather than 8-bit quantities. For example, consider:

```
static char tr_tab[256] = {...};
...
int i, ch;
...
    i = fgetc(f);   /* should be i = (unsigned char) fgetc(f) */
    ch = tr_tab[i]; /* WRONG if chars are signed... */
```

Note that declaring `i` to be unsigned int does not help (it merely causes `ch = tr_tab[i]` to index a very long way off the other end of the array!).

In non-ANSI dialects of C there is no way to explicitly declare a signed char, so plain chars tend to be signed by default (as with the ARM C compiler in `-pcc` mode). In ANSI C, a char may be plain, signed or unsigned, so a plain char tends to be whatever is most natural for the target (unsigned char on the ARM).

2.10.2 Byte ordering

A highly non-portable feature of many C programs is the implicit or explicit exploitation of byte ordering within a word of store. Such assumptions tend to arise when:

- copying objects word by word (rather than byte by byte)
- inputting and outputting binary values
- extracting bytes from, or inserting bytes into, words using a mixture of shift-and-mask and byte addressing

A contrived example which illustrates the essential pitfalls is:

```
unsigned a;
char *p = (char *)&a;
unsigned w = AN_ARBITRARY_VALUE;
while (w != 0)/* put w in a */
{   *p++ = w;/* or, maybe, w byte-reversed... */
    w >>= 8;
}
```

This code will only work on a machine with little-endian byte order.

The best solution to this class of problems is either to write code which does not rely on byte order, or to have separate code to deal appropriately with the different byte orders.

2.10.3 Store alignment

The only guarantee given in the ANSI C Standard regarding the alignment of members of a struct is that a hole (caused by padding) cannot occur at the beginning of the struct.

The values of holes created by alignment restrictions are undefined, and you should not make assumptions about these values. Strictly, two structures with identical members, each having identical values, will only be found to be equal if field-by-field comparison is used; a byte-by-byte, or word-by-word, comparison cannot be guaranteed to indicate equality.

In practice, this can be a real problem for both auto structs and structs allocated dynamically using `malloc`. If byte-by-byte comparability of such structures is required, they must be zeroed using `memset()` before assigning field values.

Padding may also have implications for the space required by a large array of structs.
For example:

```
#define ARRSIZE 10000
typedef struct
{   int i;
    short s;
} ELEM;
ELEM arr[ARRSIZE];
```

may require 40KB, 60KB or 80KB depending on the size and alignment of ints and shorts (assume a short occupies 2 bytes, 2-byte aligned; then consider a 2-byte int, a 4-byte int 2-byte aligned, and a 4-byte int 4-byte aligned). For more information, refer to *Packed structs* on page 2-29

2.10.4 Pointers and pointer arithmetic

A deficiency of the original definition of C, and of its subsequent use, has been the relatively unrestrained conversion between pointers to different data types and integers or longs. Much existing code makes the assumption that a pointer can safely be held in either a long int or an int variable. While this may be true in many implementations on many machines, it is a highly non-portable feature. Also, there is no single arithmetic type which is guaranteed to hold a pointer (long or unsigned long is probably safer than int or unsigned int).

The problem is further compounded when taking the difference of two pointers by performing a subtraction. When the difference is large, this approach is full of potential errors. ANSI C defines a type `ptrdiff_t`, which is capable of reliably storing the result of subtracting two pointer values of the same type; a typical use of this mechanism would be to apply it to pointers into the same array.

Although the difference between any two pointers of similar type may be meaningful in a flat address space, only the difference between two pointers into the same object need be meaningful in a segmented address space.

Evaluation order

Finally, there are problems of evaluation order with address arithmetic:

```
long int base, offset;
char *p1, *p2;
....
offset = base + (p2 - p1);/* intended effect */
```

If the expression were:

```
offset = (base + p2) - p1;
```

in a flat address space without holes the expressions are equivalent. In a segmented address space, (p2 - p1) may well be a valid offset within a segment, whereas (base + p2) may be an invalid address. If, in the second case, the validity is checked before subtracting p1, then the expression will fault. This latter class of problem will be familiar to MS-DOS programmers, but alien to those whose main experience is of Unix.

2.10.5 Function-argument evaluation

While the evaluation of operands to operators as ‘,’ and || is defined to be strictly left-to-right (including all side-effects), the same does not apply to function-argument evaluation. For example, in the following function call, it is unclear whether the call is `f(3, 3)` or `f(4, 3)`:

```
i = 3;
f(i, i++);
```

It is generally unwise for argument expressions to have side effects, for many reasons.

2.10.6 System-specific code

The direct use of operating system calls is obviously non-portable, though often necessary. It helps to isolate such code in target-specific modules, behind target-independent interfaces.

Filenames and filename processing are common sources of non-portability which are often difficult to deal with. Again, the best approach is to localise all such processing.

Binary data files are inherently non-portable. Often the only solution to this problem may be the use of some portable external representation.

2.11 ANSI C vs K&R C

The ISO C Standard has tightened the definition of many of the vague areas of K&R C. This results in a much clearer definition of a correct C program. However, if programs have been written to exploit particular vague features of K&R C, their results may not be as expected when porting to an ANSI C environment. In the following sections, there is a list of the major language differences between ANSI and K&R C. Library differences are discussed in a later section.

2.11.1 Lexical elements

In ANSI C, the ordering of phases of translation is well defined. Of special note is the preprocessor which is conceptually token-based (which does not yield the same results as might be expected from pure text manipulation, because the boundaries between juxtaposed tokens are visible). A number of new keywords have been introduced into ANSI C, as shown in *New ANSI C keywords* on page 2-43.

The following lexical changes have also been made:

- Each struct and union has its own distinct name space for member names.
- Suffixes U and L (or u and l), can be used to explicitly denote unsigned and long constants (eg. `32L`, `64U`, `1024UL` etc.). The U suffix is new to ANSI C.
- The use of octal constants 8 and 9 (previously defined to be octal 10 and 11 respectively) is no longer supported.
- Literal strings are considered read-only, and identical strings may be stored as one shared value (as indeed they are, by default, by the ARM C Compiler). For example:
  ```
  char *p1 = "hello";
  char *p2 = "hello";
  ```
  `p1` and `p2` will point at the same store location, where the string “hello” is held. Programs must not, therefore, modify literal strings, (beware of Unix’s `tmpnam()` and similar functions, which do this).
- Variadic functions (those which take a variable number of actual arguments) are declared explicitly using an ellipsis (...). For example:
  ```
  int printf(const char *fmt, ...);
  ```
- Empty comments `/**/` are replaced by a single space, (use the preprocessor directive ## to do token-pasting if you previously used /**/ to do this).

Note: *The K&R C practice of using* `long float` *to denote* `double` *is outlawed in ANSI C.*

Keyword	Type	Description
`volatile`	type qualifier	Means that the qualified object may be modified in ways unknown to the implementation, or that access to it may have other unknown side effects. Examples of objects correctly described as volatile include device registers, semaphores and data shared with asynchronous signal handlers. In general, expressions involving volatile objects cannot be optimised by the compiler.
`const`	type qualifier	Indicates that an object's value will not be changed by the executing program (and in some contexts permits a language system to enforce this by allocating the object in read-only store).
`void`	type specifier	Indicates a non-existent value for an expression.
`void *`	type specifier	Describes a generic pointer to or from which any pointer value can be assigned, without loss of information.
`signed`	type specifier	May be used wherever unsigned is valid (eg. to specify signed char explicitly).
`long double`		This is a new floating-point type.

Table 2-6: New ANSI C keywords

2.11.2 Arithmetic

ANSI C uses value-preserving rules for arithmetic conversions (whereas K&R C implementations tend to use unsigned-preserving rules). The following example does signed division, where unsigned-preserving implementations would do unsigned division:

```
int f(int x, unsigned char y)
{
    return (x+y)/2;
}
```

Apart from value-preserving rules, arithmetic conversions follow those of K&R C, with additional rules for long double and unsigned long int. You can now perform float arithmetic without widening to double, (note, however, that the ARM C system does not yet do this).

Floating-point values truncate towards zero when they are converted to integral types.

It is illegal to assign function pointers to data pointers and vice versa. An explicit cast must be used. The only exception to this is for the value 0, as in:

```
int (*pfi)();
pfi = 0;
```

Assignment compatibility between structs and unions is now stricter. For example:

```
struct {char a; int b;} v1;
struct {char a; int b;} v2;
v1 = v2;  /* illegal because v1 and v2 have different types */
```

To the compiler, v1 and v2 have different types because they have not been declared with the same tag. You can do this correctly as follows:

```
struct mytag {char a; int b;};
struct mytag v1;
struct mytag v2;
v1 = v2;
```

2.11.3 Expressions

Structs and unions may be passed by value as arguments to functions.

Given a pointer to a function declared as for example, `int (*pfi)()`, the function to which it points can be called either by `pfi()`; or `(*pfi)()`.

Because of the use of distinct name spaces for struct and union members, absolute machine addresses must be explicitly cast before being used as struct or union pointers:

```
((struct io_space *)0x00ff)->io_buf;
```

2.11.4 Declarations

Perhaps the greatest impact on C of the ISO Standard has been the adoption of function prototypes. A function prototype declares the return type and argument types of a function. The following example declares a function returning int, with one int and one float argument:

```
int f(int, float);
```

This means that a function’s argument types are part of the type of the function, giving the advantage of stricter type-checking, especially between separately-compiled source files.

A function definition (which is also a prototype) is similar except that identifiers must be given for the arguments, for example, `int f(int i, float f)`. You can still use old-style function declarations and definitions, but it is advisable to convert to the new style. It is also possible to mix old and new styles of function declaration. If the function declaration which is in scope is an old style one, normal integral promotions are performed for integral arguments, and floats are converted to double. If the function declaration which is in scope is a new-style one, arguments are converted as in normal assignment statements.

Empty declarations are now illegal.

Arrays cannot be defined to have zero or negative size.

2.11.5 Statements

ANSI has defined the minimum attributes of control statements. For example:

- the minimum number of case limbs which must be supported by a compiler
- the minimum nesting of control constructs

These minimum values are not particularly generous and may prove troublesome if highly portable code is required.

In general, the only limit imposed by the ARM C compiler is that of available memory. A future release may support an option to warn if any of the ANSI-guaranteed limits are violated.

A value returned from `main()` is guaranteed to be used as the program's exit code.

Values used in the controlling statement and labels of a switch can be of any integral type.

2.11.6 Preprocessor

Preprocessor directives cannot be redefined.

There is a new ## directive for token-pasting.

There is a directive # which produces a string literal from its following characters. This is useful when you want to embed a macro argument in a string.

The C compiler allows use of C++ style to introduce comments except in `_strict` mode.

The order of phases of translation is well defined and is as follows for the preprocessing phases:

1. Map source file characters to the source character set (this includes replacing trigraphs).
2. Delete all newline characters which are immediately preceded by \.
3. Divide the source file into preprocessing tokens and sequences of white space characters (comments are replaced by a single space).
4. Execute preprocessing directives and expand macros.

Any #include files are passed through steps 1-4 recursively.

2.11.7 Predefined macros

Macro	Value	Notes
`__STDC__`	1	defined when in ANSI mode (not for pcc)
`__arm`	1	defined if using armcc
`__thumb`	1	defined if using tcc
`__SOFTFP__`	1	defined if compiling for software floating point library (see -apcs option and ➲*Chapter 16, Software Floating Point*)
`_cplusplus`	1	defined if using armcpp
`__CLK_TCK`	100	centisecond clock definition
`__LINE__`	<line number>	as defined by ANSI
`__FILE__`	<filename string>	as defined by ANSI
`__DATE__`	<date string>	as defined by ANSI
`__TIME__`	<time string>	as defined by ANSI
`__CC_NORCROFT`	1	set by all Norcroft (Codemist) compilers

Table 2-7: Predefined macros

2.12 PCC Compatibility Mode

This section discusses the differences apparent when the compiler is used in pcc mode.

When given the `-pcc` command-line flag, the C compiler accepts (Berkeley) Unix-compatible C, as defined by the implementation of the Portable C Compiler and subject to the restrictions which are noted below.

In essence, PCC-style C is K&R C, together with a small number of extensions, and some clarifications of language features.

2.12.1 Language and preprocessor compatibility

In pcc mode, the ARM C compiler accepts K&R C, but it does not accept many of pcc's old-style compatibility features, whose use has been deprecated. The differences are:

- Compound assignment operators where the = sign comes first are accepted (with a warning) by some PCCs. An example is =+ instead of +=. ARM C does not allow this ordering of the characters in the token.
- The = sign before a static initialiser was not required by some very old C compilers. ARM C does not support this idiom.
- The following usage is found in some Unix tools pre-dating Unix Version 7:

  ```
  struct {int a, b;};
  double d;
  d.a = 0; d.b = 0x....;
  ```

This is accepted by some Unix PCCs and may cause problems when porting old code:

- Enums are less strongly typed than is usual under PCCs. Enum is an extension to K&R C which has been standardised by ANSI somewhat differently from the BSD PCC implementation.
- Chars are signed by default in pcc mode (unsigned in ANSI mode).
- In pcc mode, the compiler permits the use of the ANSI "..." notation which signifies that a variable number of formal arguments follow.
- In order to cater for PCC-style use of variadic functions, a version of the PCC header file `varargs.h` is supplied with the release.

With the exception of enums, the compiler's type checking is generally stricter than PCC, much more like lint's. The ARM C compiler attempts to strike a balance between giving too many warnings when compiling known, working code, and warning of poor or non-portable programming practices.

Many PCCs compile code which cannot execute in even a slightly different environment. ARM has tried to help those who need to port C among machines where the following varies:

- the order of bytes within a word (little-endian ARM, VAX, Intel versus big-endian Motorola, IBM370)
- the default size of int (four bytes versus two bytes in many PC implementations)
- the default size of pointers (not always the same as int)
- whether values of type char default to signed or unsigned char
- the default handling of undefined and implementation-defined aspects of the C language

The compiler's preprocessor is believed to be equivalent to a BSD Unix cpp except for the points listed below. Unfortunately, cpp is only defined by its implementation, and although equivalence has been tested over a large body of Unix source code, completely identical behaviour cannot be guaranteed. Some of the points listed below only apply when the `-E` option is used with the `cc` command:

- there is a different treatment of white space sequences (benign)
- newline is processed by `cc -E`, but passed by cpp (making lines longer than expected (`cc -E` only)
- cpp breaks long lines at a token boundary; `cc -E` does not

  This may break line-size constraints when the source is later consumed by another program (`cc -E` only).
- the handling of unrecognised directives is different (this is mostly benign)

2.12.2 Standard headers and libraries

Use of the compiler in pcc mode does not preclude the use of the standard ANSI headers built in to the compiler or the use of the run-time library supplied with the C compiler. The ANSI library does not contain the whole of the Unix C library, but it does contain many commonly used functions. However, watch for functions with different names, or a slightly different definition, or those in different standard places. Unless you direct otherwise using `-j`, the C compiler will attempt to satisfy references to `stdio.h` from its built-in filing system, for example.

Listed below are a number of differences between the ANSI C Library and the BSD Unix library. They are placed under headings corresponding to the ANSI header files:

`ctype.h`	There are no `isascii()` and `toascii()` functions, since ANSI C is not character-set specific.
`errno.h`	On BSD systems `sys_nerr` and `sys_errlist()` are defined to give error messages corresponding to error numbers. ANSI C does not have these, but provides similar functionality via `perror(const char *s)`. This displays the string pointed to by `s` followed by a system error message corresponding to the current value of `errno`. There is also `char *strerror(int errnum)` which, when given a value of `errno`, returns its textual

	equivalent.
`math.h`	The #defined value `HUGE`, found in BSD libraries, is called `HUGE_VAL` in ANSI C. ANSI C does not have `asinh()`, `acosh()` or `atanh()`.
`signal.h`	In ANSI C the `signal()` function's prototype is:

```
extern void (*signal(int, void(*func)(int)))(int);
```

`signal()` therefore expects its second argument to be a pointer to a function returning void with one int argument. In BSD-style programs it is common to use a function returning int as a signal handler. The PCC-style function definitions shown below therefore produce a compiler warning about an implicit cast between different function pointers (since `f()` defaults to `int f()`). This is just a warning, and the correct code is generated.

```
f(signo)
int signo;
...
main()
{   extern f();
    signal(SIGINT, f);
...
```

`stdio.h`	`sprintf()` returns the number of characters printed (following Unix System V), whereas the BSD's `sprintf()` returns a pointer to the start of the character buffer. The BSD functions `ecvt()`, `fcvt()` and `gcvt()` are not included in ANSI C, as their functionality is provided by `sprintf()`
`string.h`	On BSD systems, string manipulation functions are found in `strings.h` whereas ANSI C places them in `string.h`. The ARM C Compiler also recognises `strings.h`, for PCC-compatibility. The BSD functions `index()` and `rindex()` are replaced by the ANSI functions `strchr()` and `strrchr()` respectively. Functions that refer to string lengths (and other sizes) now use ANSI type `size_t`, which in this implementation is unsigned int.
`stdlib.h`	`malloc()` has type `void *`, rather than the `char *` of the BSD `malloc()`.
`float.h`	A new header added by ANSI, giving details of floating-point precision etc.
`limits.h`	A new header added by ANSI, to give maximum and minimum limit values for integer data types.
`locale.h`	A new header added by ANSI, to provide local environment-specific features.

2.13 Machine-Specific Features

2.13.1 Pragma directives

Pragmas are recognised by the compiler in two forms:

```
#pragma -LetterOptional digit
#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 nocheck_stack

#pragma -p2
#pragma profile_statements
```

The list of recognised pragmas is shown in *Pragmas* on page 2-51. The default setting is marked with *.

2.13.2 Specifying pragmas from the command line

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

```
-zpLetterDigit
```

Certain pragmas give more local control over what can be controlled per compilation unit, from the command line. For example:

Pragma name	**Command line form**
`nowarn_implicit_fn_decls`	`-Wf`
`nowarn_deprecated`	`-Wd`
`profile`	`-p`
`profile_statements`	`-px`

2.13.3 Pragmas controlling the preprocessor

`continue_after_hash_error`	Implements a #warning "..." preprocessor directive.
`include_only_once`	Asserts that the containing `#include` file is included only once, and that if its name recurs in a subsequent `#include` directive, the directive is ignored.
`force_top_level`	Asserts that 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 the body of a function.

The values marked with a * are the default values.

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

Pragma Name	Short Form	‘no’ Form
`warn_implicit_fn_decls`	`a1 *`	`a0`
`check_memory_accesses†`	`c1`	`c0 *`
`warn_deprecated`	`d1 *`	`d0`
`continue_after_hash_error`	`e1`	`e0 *`
`FP register variable†`	`f1-f4`	`f0 *`
`include_only_once`	`i1`	`i0 *`
`optimise_crossjump`	`j1 *`	`j0`
`optimise_multiple_loads`	`m1 *`	`m0`
`profile†`	`p1`	`p0 *`
`profile_statements†`	`p2`	`p0 *`
`integer register variable`	`r1-r7`	`r0 *`
`check_stack`	`s0 *`	`s1`
`force_top_level`	`t1`	`t0 *`
`check_printf_formats`	`v1`	`v0 *`
`check_scanf_formats`	`v2`	`v0 *`
`side_effects`	`y0 *`	`y1`
`optimise_cse`	`z1 *`	`z0`

Table 2-8: Pragmas

2.13.4 Pragmas controlling printf/scanf argument checking

Pragmas `check_printf_formats` and `check_scanf_formats` control whether the actual arguments to `printf` and `scanf`, respectively, 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.

2.13.5 Pragmas controlling optimisation

Pragmas `optimise_crossjump`, `optimise_multiple_loads` and `optimise_cse` give fine control over where these optimisations are applied. For example, it is sometimes advantageous to disable cross-jumping (the 'common tail' optimisation) in the critical loop of an interpreter; and it may be helpful in a timing loop to disable common subexpression elimination and the optimisation 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 subexpression elimination. `Pragma noside_effects` asserts that the following function declarations (until the next `#pragma side_effects`) describe pure functions, invocations of which can be CSEs. See also ➲*__pure* on page 2-54.

2.13.6 Pragmas controlling code generation

Stack-limit checking

If the compiler is configured to compile code for the explicit stack limit variant of the ARM Procedure Call Standard (documented in ➲*Chapter 19, 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 to turning off this check: it typically costs only two instructions and two machine cycles per function call. The one circumstance where `nocheck_stack` must be used is 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 would result 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   (?=1,2,4 for byte, short, long reads, respectively)
__rt_wr?chk   (?=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 *Registers* on page 2-28), 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 5 and 7 integer registers (v1-v7, machine registers R4-R10) and 4 floating point registers (F4-F7) are available as register variables. In practice it is probably unwise to use more than 3 global integer register variables and 2 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 link-time check 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 2-55.

2.13.7 Special function declaration keywords

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

Note: *None of these 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 `armcc` 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 (excluding 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.

__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.

__swi and __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 a 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 19, ARM Procedure Call Standard* for details of the default way in which structures are passed and returned.

2.13.8 Special 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 ➲*Pragmas controlling code generation* on page 2-52.

__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 ➲*Pragmas controlling code generation* on page 2-52.

Note: The global register, whether specified by keyword or pragmas, must be specified in all declarations of the same variable. Thus

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

is an error.

2.14 Floating Point Support

By default, armcc and tcc generate calls to floating-point library functions. (see *Chapter 16, Software Floating Point*). armcc can also be switched to generate ARM floating-point coprocessor instructions (see *Chapter 19, ARM Procedure Call Standard*).

The ARM's floating-point instruction set is supported either by an attached floating-point coprocessor (hardware coprocessors 1 and 2) or by an instruction emulator entered from the undefined instruction trap.

Normally the floating-point instruction emulator is installed by the environment in which the program is executing. However, for a completely standalone application, the program can install the floating-point emulator itself.

Floating-point emulator

The ARM Floating Point instruction set Emulator (FPE) is supplied with the ARM C system as a linkable object file. Its environmental dependencies are all via a *stub*, supplied as an assembly language source. This stub file, `fpestub`, documents how to attach an FPE to the invalid instruction trap location (address 0x4).

It is intended that the FPE and the `fpestub` be linked together with whatever else is required to make a standalone module on the target hardware. The `fpestub` contains two entries for:

initialisation	Attaching it to the invalid instruction trap vector. This should be called on activation of the standalone module.
finalisation	Removing it from the invalid instruction trap vector. This should be called on deactivation of the standalone module.

For testing purposes, the FPE, `fpestub` and a test application can be linked together to make a single, standalone application. The application must call `__fp_initialise` before using any floating point instructions, and `__fp_finalise` before exiting.

Note: *For standalone applications, it is better to use the software floating-point library, as it is faster and the generated ARM code only includes the fp routines that are actually used.*

Thumb: The Thumb C compiler does not generate floating-point instructions as there are no floating-point instructions available in Thumb.

2.15 ARM/Thumb interworking

2.15.1 Introduction

Code compiled (or assembled) for ARM and Thumb can be freely mixed providing the code conforms to the ARM Procedure Call Standard and Thumb Procedure Call Standard respectively.

Compiled code automatically conforms to these standards. Assembler programmers must ensure their code conforms to these standards.

The ARM linker automatically detects when you are mixing ARM and Thumb code and generates small code segments called *veneers*, These veneers perform an ARM-Thumb state change on function entry and exit when an ARM function is called from Thumb state and vice-versa.

2.15.2 Compiling code for interworking

When interworking ARM and Thumb code, code may be compiled specially for interworking using the `-apcs 3/interwork` option on both the ARM and Thumb compilers. Alternatively it may be compiled as normal.

In both cases, interworking between ARM and Thumb will work. However, the following trade-offs must be made when deciding whether to compile code specially for interworking or not.

Code not compiled for interworking:

- does not support ARM/Thumb interercalling via function pointers (ie. only direct calls between ARM and Thumb are supported)
- uses larger veneers than code compiled for interworking (the veneers are 20 bytes per ARM/Thumb caller/called routine pair)

Code compiled for interworking:

- generates slightly larger code for Thumb (typically 1% larger) and marginally larger code for ARM

  This will have a correspondingly small effect on performance.
- cannot be used on non-Thumb ARMs
- allows intercalling using function pointers
- uses smaller veneers (8 or 12 bytes per called routine)

The trade-off depends on the number of intercalls made. If only a few direct intercalls are made, it is best not to compile the code for interworking.

As the number of intercalls increases the larger size of the ARM/Thumb veneers will start to dominate so it is better to compile the code for interworking.

2.15.3 Mixing interworking/non-interworking code

It is also possible to mix code compiled for interworking with code not compiled for interworking. You may wish to do this if sections of your code perform a large amount of ARM/Thumb interworking whereas other sections are compiled solely in ARM or Thumb.

When mixing interworking and non-interworking code, the type of called routine determines which style of interworking veneer should be used. Therefore you can compile any module containing an ARM/Thumb called routine with interworking and all other modules without interworking.

To help determine which routines are intercalled, the linker generates a warning when it detects a direct ARM/Thumb intercall where the called routine is not compiled for interworking.

The following warnings may be generated:

```
Interworking call from ARM to Thumb code symbol symbol in object(area)
Interworking call from Thumb to ARM code symbol symbol in object(area)
```

These indicate that an ARM to Thumb or Thumb to ARM intercall respectively has been detected from the object module *`object`* to the routine *`symbol`*. The module containing *`symbol`* may then be compiled with interworking.

2.15.4 Interworking code using function pointers

To interwork code which uses function pointers both caller and called routine must be compiled with interworking for the code to work at all. No warning can be issued if an attempt is made to interwork function pointer calls which have been compiled without interworking, however the call will fail at run time.

2.15.5 Using two copies of the same function

Occasionally you may wish to include two functions of the same name, one compiled in ARM the other compiled in Thumb. Normally the linker will give an error if there is a duplicate definition of a symbol, however the linker will allow duplicate definition provided each definition is of a different type (ie. one definition defines a Thumb routine, the other defines and ARM routine).

You may wish to do this, for example, if you have a speed critical routine in a system with both 16-bit and 32-bit memory where the overhead of the interworking veneer would degrade the performance or where the ARM or Thumb version of the routine may not be loaded due to the use of an overlay scheme.

The linker will generate the following warning when two versions of the same routine are included:

```
Both ARM & Thumb versions of symbol present in image
```

This is a warning intended to advise you in case you were accidentally including two copies of the same routine. If this is what you intended the warning may be ignored.

Note: *When both versions of a routine are present in an image and a call is made via a function pointer, it is not possible to determine which version of the routine will be called.*

2.15.6 The C library

Two variants of the ARM C libraries are provided, one set which has been compiled for interworking (`armlib_i.32l` and `armlib_i.32b`) and one which has not been compiled for interworking (`armlib.32l` and `armlib.32b`).

Two variants of the Thumb C library are also provided, (`armlib_i.16l` and `armlib_i.16b`) for interworking and (`armlib.16l` and `armlib.16b`) for non-interworking.

The non-interworking variants of the ARM C libraries (`armlib.32l` and `armlib.32b`) must be used when creating code to run on a non-Thumb ARM.

The linker has an option to allow forcible inclusion of specific modules from a library. You may wish to do this if you need to select the ARM or Thumb version of a standard C library routine explicitly. You may also use this to forcibly include both ARM and Thumb versions of a routine.

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

Examples

Forcibly use the ARM version of `strlen` and take all other routines from the Thumb library.

```
armlink -o myprog myprog.o armlib.32l(strlen.o) armlib.16l
```

Forcibly include both ARM and Thumb versions of all `str...` functions and take all other routines from the Thumb library.

```
armlink -o myprog myprog.o armlib.16l(str*) armlib.32l(str*) armlib.16l
```

Note: *Depending on the command shell you are using you may need to quote the character (,) and * to enter them on the command line.*

2.15.7 Example

Compile the following code segments:

```
armcc -c -li -apcs 3/noswst sum32.c
armcc -c -li -apcs 3/noswst init32.c
tcc -c -li sum16.c
tcc -c -li init16.c
tcc -c -li main.c
```

Link it:

```
tcc -o -li test sum16.o sum32.o init16.o init32.o main.o
```

The linker will produce the following warnings:

```
ARM Linker: (Warning) Both ARM & Thumb versions of checksum present in
image.
```

```
ARM Linker: (Warning) Interworking call from Thumb to ARM code symbol
init32 in main.o(C$$code).
```

When you run it using armsd, you should get the following output:

```
Init16 called
Calculating checksum of 0x20000 bytes of 16 bit memory
Checksum = eaea
Init32 called
Calculating checksum of 0x8000 bytes of 32 bit memory
Checksum = cbcbcbcb
```

sum16.c

```
#include <stdio.h>
unsigned int checksum(unsigned short *memory_base, int n)
{
          unsigned int sum;

          printf("Calculating checksum of 0x%x bytes of 16 bit
          memory\n", n);
          sum = 0;
          n /= sizeof(unsigned short);
          while (--n >= 0) sum ^= *memory_base++;
          return sum;
}
```

sum32.c

```
#include <stdio.h>
unsigned int checksum(unsigned int *memory_base, int n)
{
          unsigned int sum;

          printf("Calculating checksum of 0x%x bytes of 32 bit
          memory\n", n);
          sum = 0;
          n /= sizeof(unsigned int);
          while (--n >= 0) sum ^= *memory_base++;
          return sum;
}
```

init16.c

```
#include <stdio.h>
/* Dummy memory map for example - 64K 16 bit memory */
static unsigned short memory[0x10000];
extern unsigned int checksum(unsigned short *, int n);
void init16(void)
{
```

```
          printf("Init16 called\n");

          /* Initialise 1st memory word to bit pattern */
          memory[0] = 0xEAEA;

          /* Checksum should be 0xEAEA */
          printf("Checksum = %x\n", checksum(memory, sizeof(memory)));

          /* ... rest of initialisation */
}
```

init32.c

```
#include <stdio.h>
/* Dummy memory map for example - 8K 32 bit memory */
static unsigned int memory[0x2000];
extern unsigned int checksum(unsigned int *, int n);
void init32(void)
{
          printf("Init32 called\n");

          /* Initialise 1st memory word to bit pattern */
          memory[0] = 0xCBCBCBCB;

          /* Checksum should be 0xCBCBCBCB */
          printf("Checksum = %x\n", checksum(memory, sizeof(memory)));

          /* ... rest of initialisation */
}
```

main.c

```
extern void init16(void);
extern void init32(void);
int main(void)
{
          init16();
          init32();
}
```

3 Assembler

This chapter describes the ARM Assembler.

Assembler

3.1 Overview

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.

For more information about ARM Assembly Language see *Chapter 4, ARM Instruction Set*.

For more information about Thumb Assembly Language see *Chapter 5, Thumb Instruction Set*.

For more information about linking, see *Chapter 6, Linker*.

armasm is a two-pass assembler, processing its source files twice to reduce the amount of internal state that it needs to keep. This affects the user in a few ways which are discussed below.

3.2 Command Line Options

The command to invoke armasm or tasm takes either of the forms:

```
toolname {options} sourcefile objectfile
toolname {options} -o objectfile sourcefile
```

where `toolname` is either `armasm` or `tasm`.

The options are listed below. Upper-case is used to show the allowable abbreviations.

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

`-NOTerse`	Turns the *terse* flag off (the default is on). 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.
`-WIdth` *n*	Sets listing page width (the default is 79).
`-Length` *n*	Sets listing page length (the default is 66). Setting the length to zero produces an unpaged listing.
`-Xref`	Lists cross-referencing information on symbols; where they were defined and where they were used, both inside and outside macros. Default is off.

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

`-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 being the directory where the current file was found.
`-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"`.
`-NOCache`	Turns off source caching, (the default is on). Source caching is performed when reading source files on the first pass, so that they can be read from memory during the second pass.
`-MaxCache` *`n`*	Sets the maximum source cache size. The default is 8MB.
`-NOEsc`	Ignore C-style special characters ('\n', '\t' etc).
`-noregs`	Tells the assembler not to predefine the implicit register names (R0-R15, F0-F7, a1-a4, v1-v6, sl, fp, ip, sp, lr, pc)
`-NOWarn`	Turns off warning messages.
`-g`	Outputs ASD debugging tables, suitable for use with armsd.
`-Errors` *`errorfile`*	Output error messages to *`errorfile`*.
`-LIttleend`	Assemble code suitable for a little-endian ARM, (by setting the built-in variable {ENDIAN} to “little”).
`-BIgend`	Assemble code suitable for a big-endian ARM, (by setting the built-in variable {ENDIAN} to “big”).
`-CPU` *`ARMcore`*	Set target ARM core to *`ARMcore`*. Currently this can take the values `ARM6`, `ARM7`, `ARM7M`, `ARM7TM`, `ARM8`. Some processor-specific instructions will produce warnings if assembled for the wrong ARM core.
`-ARCH` *`architecture`*	Set the target architecture. Legitimate values are 3, 3M, 4, 4T. Some processor-specific instructions will produce either errors or warnings if assembled for the wrong target architecture.
`-unsafe`	Change any errors produced due to selected architecture and cpu into warnings.
`-CheckReglist`	Check 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.
`-VIA` *`file`*	*`file`* is opened and more armasm command line arguments are read in from it. This is intended mainly for hostings such as

	the PC where command line length is severely limited.
`-Help`	Displays a summary of the command-line options.
`-Apcs` *`option`*`{/`*`qualifier`*`}{/`*`qualifier`*`...}`	Specifies whether the ARM Procedure Call Standard is in use, and also specifies some attributes of CODE AREAs. See the following subsection for more information on APCS options.

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.

APCS options

There are two APCS options:

- `NONE`
- `3`

Qualifiers should only be used with 3.

Predeclared register names

By default the following register names are predeclared:

- R0-15
- r0-15
- sp and SP
- lr and LR
- pc and PC

If the APCS is in use the following register names are also pre-declared:

- a1-a4
- v1-v6
- sl
- fp, ip, and sp

The qualifiers are as follows:

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

`/32bit`	Is the default setting and 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.
`/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 of 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 what 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 reentrant, v7 if not.

3.3 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, or a mixture of both.

This section deals with features that are common to both ARM and Thumb assembly language. For language-specific information, see *Chapter 4, ARM Instruction Set* and *Chapter 5, Thumb Instruction Set*.

3.3.1 Case rules

Instruction mnemonics and register names may be written in upper or lower case (but not mixed). Directives must be written in upper case.

3.3.2 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 are allowed to 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 + end of line sequence is treated by armasm as white space.

Note: *The backslash + end of line sequence should not be used within quoted strings.*

3.3.3 AREAs

AREAs are the 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, an assembly, or the output of a compilation, consists of two AREAs, one for the code (usually marked read-only), and one for the data which may be written to. A reentrant object will generally have a third AREA marked BASED sb (see below), which will contain relocatable address constants. This allows the code area to be read-only, position-independent and reentrant, making it easily ROM-able.

In ARM assembly language, each AREA begins with an AREA directive. If the directive is missing the assembler will generate an AREA with an unlikely name (|$$$$$$$|) and produce a diagnostic message to this effect. This will limit the number of spurious errors caused by the missing directive, but will not lead to a successful assembly.

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`	Absolute: rooted at a fixed address.
`REL`	Relocatable: may be relocated by the Linker (the default).
`PIC`	Position Independent Code: will execute where loaded without modification.
`CODE`	Contains machine instructions.
`DATA`	Contains data, not instructions.
`READONLY`	This area will not be written to.
`COMDEF`	Common area definition.
`COMMON`	Common area.
`NOINIT`	Data AREA initialised to zero: contains only space reservation directives, with no initialised values.
`REENTRANT`	The code AREA is reentrant.
`HALFWORD`	The code AREA containsARM halfword instructions.
`INTERWORK`	The code AREA is suitable for ARM/Thumb interworking.
`BASED` *Rn*	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 *Chapter 19, ARM Procedure Call Standard*.
`ALIGN=`*expression*	The ALIGN sub-directive 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 12 inclusive.

3.3.4 ORG and ABS

The ORG (origin) directive is used to set 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 will create objects which cannot be linked. In general it only makes sense to use ORG in programs consisting of one AREA, which need to map fixed hardware addresses such as trap vector locations. Otherwise ORG should be avoided.

3.3.5 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 *3.5.2 Local and global variables—GBL, LCL and SET* on page 3-18). Addresses are assigned by the Assembler as assembly proceeds, some remaining in symbolic, relocatable form until link time.

- Symbols must start with a letter in either upper or lower case; the assembler is case-sensitive and 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 the uses of the term through their relative positions in the input line, a programmer may not always be able to do so.
- Symbol length is limited by the 255 character line length limit.

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.

3.3.6 Labels

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

3.3.7 Local labels

The local label, a subclass of label, begins with a number in the range 0-99. Local labels work in conjunction with the ROUT directive and are most 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

To begin a local label area, use:

```
{label} ROUT
```

The label area will start with the next line of source, and will end with the next ROUT directive or the end of the program.

Defining local labels

Local labels are defined as:

```
number{routinename}
```

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

A reference to a local label has the following syntax:

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

%	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; use B for backward or F for 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 to look at all macro levels, T to look only at this macro level, or, if *y* is absent, to look at all macro from the current level to the top level.
n	Is the number of the local label.
routinename	Is optional, but if present will be checked against the enclosing ROUT's label.

3.3.8 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.

3.3.9 Constants

Numbers	Numeric constants are accepted in three forms: decimal (for example `123`), hexadecimal (eg. `&7B`), and *n*_*xxx*, where *n* is a base between 2 and 9 and *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 of an “escaped” character, using the standard C escape characters.

3.3.10 The END directive

Every assembly language source must end with:

```
END
```

on a line by itself.

3.4 Directives

3.4.1 Storage reservation and initialisation–DCB, DCW and DCD

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

The syntax of the first three directives 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, the characters of which 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 thus:

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

The syntax of % is:

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

This directive will initialise to zero the number of bytes specified by the *`numeric expression`*.

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

3.4.2 Floating point store initialisation–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{-}integereg. 1E3, -4E-9
{-}{integer}.integer{E{-}integer}eg. 1.0, -.1, 3.1E6
```

E may also be written in lower case.

3.4.3 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
```

The ^ 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. @ 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]
```

3.4.4 Organisational directives—END, ORG, LTORG and KEEP

`END`

The assembler stops processing a source file when it reaches the END directive. If assembly of the file was invoked by a GET directive, the assembler returns and continues after the GET directive (see section *3.4.6 Links to other source files–GET/INCLUDE* on page 3-13). If END is reached in the top-level source file during the first pass without any errors, the second pass will begin. Failing to end a file with END is an error.

`ORG` *`numeric-expression`*

A program's origin is determined by the ORG directive, which sets the initial value of the program location counter. Only one ORG is allowed in an assembly and no ARM instructions or store initialisation directives may precede it. If there is no ORG, the program is relocatable and the program counter is initialised to 0.

```
LTORG
```

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

```
KEEP {symbol}
```

The assembler does not by default describe local (non-exported), symbols in its output object file (see ➲*3.4.5 Links to other object files–IMPORT and EXPORT*). However, they can be retained in the object file's symbol table by using the KEEP directive. If the directive is used alone all symbols are kept; if only a specific symbol needs to be kept it can be specified by name.

3.4.5 Links to other object files–IMPORT and EXPORT

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

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

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

EXPORT declares a symbol for use at link time by other, separate object files. FPREGARGS signifies that the symbol defines a function which expects floating point arguments to be passed to it in floating point registers. DATA denotes that the symbol defines a code-segment datum rather than a function or a procedure entry point, and LEAF that it is a leaf function which calls no other functions.

3.4.6 Links to other source files–GET/INCLUDE

`GET` *`filename`*

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

`INCLUDE` *`filename`*

INCLUDE is a synonym for GET.

3.4.7 Diagnostic generation–ASSERT and !

`ASSERT` *`logical-expression`*

ASSERT 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.

3.4.8 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 pass one listing. These settings can be altered by adding the appropriate values from the list below, and using them with the OPT directive as follows:

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 pass one listing.
2048	Turns off the pass one 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.

3.4.9 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
```

3.4.10 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 then encountered, the assembler will insert 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 syntax of this directive is:

```
label RLIST list-of-registers
```

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

If the `-CheckReglist` command-line option is selected then 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 {}. 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.

3.4.11 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 to which it is attached is a ‘data-in-code’ label (ie. it defines an area of data within a code segment). This directive *must* be specified if you are defining data in a code area.

3.5 Symbolic Capabilities

3.5.1 Setting constants

The EQU and * directives are used to give a symbolic name to a fixed or program-relative value. The syntax is:

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

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

The FN directive 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
```

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

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

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

3.5.2 Local and global variables—GBL, LCL and SET

While most symbols have fixed values determined during assembly, variables have values which may change as assembly proceeds. The assembler supports both global and local variables. 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 section ➲*3.9 Macros* on page 3-25). Variables must be declared before use with one of these directives.

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 initialises a local arithmetic variable (initial state zero)
LCLL	declares and initialises a local logical variable (initial state false)
LCLS	declares and initialises a local string variable (initial state null string)

The syntax of these directives is:

```
directive variable-name
```

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.

3.5.3 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 *3.6.1 Unary operators* on page 3-21), string variables by their value.

3.5.4 Built-in variables

There are seven 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 has the value 32
`{CPU}`	Has the value “generic ARM” if no CPU has been specified, or the name of the selected cpu if one has.

`{ARCHITECTURE}`	Has the value of the selected ARM architecture: one of 3, 3M, 4, 4T
`{PCSTOREOFFSET}`	Is the offset between the address of an STR PC,[...] or STM Rb,{... PC} instruction and the value of PC stored out. This varies depending on the CPU and architecture specified.

3.6 Expressions and Operators

Expressions are combinations of simple values, unary and binary operators, and brackets. There is a strict order of precedence in their evaluation: expressions in brackets are evaluated first, then 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.

3.6.1 Unary operators

Unary operators have the highest precedence (bind most tightly) and so 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`	`:BASE:A`	If A is a PC-relative or register-relative expression, BASE returns the number of its register component and INDEX the offset from that base register.
`INDEX`	`:INDEX:A`	BASE and INDEX are most likely to be of use within macros.
`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.
+	+A	Unary plus.
–	–A	Unary negate. + and – can act on numeric, program-relative and string expressions.
`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 3-1: Operator Precedence

3.6.2 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:

`*`	`A*B`	multiply
`/`	`A/B`	divide
`MOD`	`A:MOD:B`	A modulo B

These operators act only on numeric expressions.

String manipulation operators

`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

In the two slicing operators LEFT and RIGHT, A must be a string and B must be a numeric expression.

Shift operators

`ROL`	`A:ROL:B`	rotate A left B bits
`ROR`	`A:ROR:B`	rotate A right B bits
`SHL`	`A:SHL:B`	shift A left B bits
`SHR`	`A:SHR:B`	shift A right B bits

The shift operators act on numeric expressions, shifting or rotating the first operand by the amount specified by the second. Note that SHR is a logical shift and does not propagate the sign bit.

Addition and logical operators

`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

The bitwise operators act on numeric expressions. The operation is performed independently on each bit of the operands to produce the result.

Relational operators

`=`	`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

The relational operators act upon two operands of the same type to produce a logical value. Allowable types of operand are numeric, program-relative, register-relative, and 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}.

Boolean operators

These are the weakest binding operators with the lowest precedence.

`LAND`	`A:LAND:B`	logical AND of A and B
`LOR`	`A:LOR:B`	logical OR of A and B
`LEOR`	`A:LEOR:B`	ogical Exclusive OR of A and B

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

3.7 Conditional Assembly—[, | and]

Sections of a source file may be assembled conditionally, only if certain conditions are true. The [and] (if and endif) directives are used to mark their start and finish; | provides an else construct. The syntax is:

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

Note that [, | and] may not be the first character of a line. If the logical-expression is true, the section will be assembled; if it is false, the second piece of code, the beginning of which is marked by | and the end of which is marked 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.

The directives IF, ELSE and ENDIF may be used instead of [, | and] respectively.

3.8 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 run-time 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
```

3.9 Macros

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

3.9.1 Defining a macro

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

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

The directive MACRO 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 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*, $*parameter*, etc., can be used in the same way as any other variables (see *3.5.2 Local and global variables—GBL, LCL and SET* on page 3-18, and *3.5.3 Variable substitution—$* on page 3-19). 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 recognise 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 un-closed 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.

3.9.2 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"}
```

3.9.3 Macro invocation

A macro defined with a pattern such as:

```
$lab   xxxx $arg1,$arg2=5,$arg3
```

can be invoked as:

```
Label  xxxx 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.

4

ARM Instruction Set

This chapter describes the ARM instruction set.

4.1 The ARM Instruction Set—Overview

The ARM instruction set may be subject to processor-specific restrictions and changes. Particular combinations of instructions must be avoided where noted, as unpredictable results may otherwise occur. Refer to the appropriate ARM processor data sheet for a precise definition of the instruction set, and also refer to companion application notes for information on relevant restrictions and changes. The most significant variations are those between ARM processors with 26- and 32-bit program counters.

4.1.1 Instruction summary

Mnemonic	Instruction	Action	See Section:
`ADC`	Add with carry	Rd := Rn + Op2 + Carry	4.3
`ADD`	Add	Rd := Rn + Op2	4.3
`AND`	AND	Rd := Rn AND Op2	4.3
`B`	Branch	R15 := address	4.2
`BIC`	Bit Clear	Rd := Rn AND NOT Op2	4.3
`BL`	Branch with Link	R14 := R15, R15 := address	4.2
`BX`	Branch and Exchange	R15 := Rn, T bit := Rn[0]	4.11
`CDP`	Coprocessor Data Processing	(Coprocessor-specific)	4.14
`CMN`	Compare Negative	CPSR flags := Rn + Op2	4.3
`CMP`	Compare	CPSR flags := Rn - Op2	4.3
`EOR`	Exclusive OR	Rd := (Rn AND NOT Op2) OR (op2 AND NOT Rn)	4.3
`LDC`	Load coprocessor from memory	Coprocessor load	4.14
`LDM`	Load multiple registers	Stack manipulation (Pop) and block copy	4.7
`LDR`	Load register from memory	Rd := (address)	4.5
`MCR`	Move CPU register to coprocessor register	cRn := rRn {<op>cRm}	4.14
`MLA`	Multiply Accumulate	Rd := (Rm * Rs) + Rn	4.8

Table 4-1: The ARM instruction set

Mnemonic	Instruction	Action	See Section:
MOV	Move register or constant	Rd : = Op2	4.3
MRC	Move from coprocessor register to CPU register	Rn := cRn {<op>cRm}	4.14
MRS	Move PSR status/flags to register	Rn := PSR	4.4
MSR	Move register to PSR status/flags	PSR := Rm	4.4
MUL	Multiply	Rd := Rm * Rs	4.8
MVN	Move negative register	Rd := 0xFFFFFFFF EOR Op2	4.3
ORR	OR	Rd := Rn OR Op2	4.3
RSB	Reverse Subtract	Rd := Op2 - Rn	4.3
RSC	Reverse Subtract with Carry	Rd := Op2 - Rn - 1 + Carry	4.3
SBC	Subtract with Carry	Rd := Rn - Op2 - 1 + Carry	4.3
SMLAL	Signed multiply accumulate long	RdHi:=signed(Rm*Rs)+RdHi+ CarryFrom((Rm*Rs)[31:0]+RdLo))	4.8
SMULL	Signed multiply long	RdHi:= signed(Rm*Rs)[63:32] RdLo:= signed(Rm*Rs)[31:0]	4.8
STC	Store coprocessor register to memory	address := CRn	4.14
STM	Store Multiple	Stack manipulation (Push) and block copy	4.7
STR	Store register to memory	<address> := Rd	4.5
SUB	Subtract	Rd := Rn - Op2	4.3
SWI	Software Interrupt	OS call	4.12
SWP	Swap register with memory	Rd := [Rn], [Rn] := Rm	4.10
TEQ	Test bitwise equality	CPSR flags := Rn EOR Op2	4.3
TST	Test bits	CPSR flags := Rn AND Op2	4.3

Table 4-1: The ARM instruction set (Continued)

Mnemonic	Instruction	Action	See Section:

`UMLAL`	Unsigned multiply accumulate long	RdLo:=(Rm*Rs)+RdLo
		RdHi:=(Rm*Rs)+RdHi+
		CarryFrom((Rm*Rs)[31:0]+RdLo))
			4.8

`UMULL`	Unsigned multiply long	RdHi:= (Rm*Rs)[63:32]
		RdLo:= (Rm*Rs)[31:0]
			4.8

Table 4-1: The ARM instruction set (Continued)

4.1.2 Conditional execution and the ‘S’ bit

All ARM instructions are conditional and are only executed if their condition field matches the N, Z, C and V condition flags of the program status register (PSR). The default condition field setting is “execute always”; other conditions are specified by appending a two-character condition mnemonic to the instruction mnemonic. These available mnemonics are shown in ➲*Table 4-2: Conditions*. A conditionally executed sequence of instructions will usually be shorter and sometimes even faster than a branched-around sequence, because it will not cause breaks in the CPU pipeline.

Mnemonic	Condition	CPU condition flags
`EQ`	EQual	Z set
`NE`	Not Equal	Z clear
`CS`	Carry Set/unsigned Higher or Same	C set
`CC`	Carry Clear/unsigned LOwer than	C clear
`MI`	Negative (MInus)	N set
`PL`	Positive (PLus)	N clear
`VS`	oVerflow Set	V set
`VC`	oVerflow Clear	V clear
`HI`	HIgher unsigned	C set and Z clear
`LS`	Lower or Same unsigned	C clear or Z set
`GE`	Greater than or Equal to	(N and V) set or (Nand V) clear
`LT`	Less Than	(N set and V clear) or (N clear and V set)
`GT`	Greater Than	((N and V) set or clear) and Z clear
`LE`	Less Than or equal to	(N set and V clear) or (N clear and V set) or Z set

Table 4-2: Conditions

`HS` (Higher or Same) and `LO` (LOwer than) are synonyms for `CS` and `CC` respectively.

Condition flags are set by executed ALU instructions and multiplies that have the ‘S’ bit set, and by executed comparison instructions. The S bit is set by appending ‘S’ to the instruction mnemonic.

4.1.3 Register names and '.'

Fifteen registers (R0 to R14), the program counter (PC), and the processor status register (PSR) are all directly accessible to the programmer. Register R15 contains the PC, and in 26-bit address ARMs it also contains the PSR. In 32-bit address ARMs, the PSR is separate, and is manipulated by separate instructions.

R14 is used as the subroutine link register, saving a copy of R15 when a Branch with Link instruction is executed (see *4.2 Branch Instructions—B and BL* on page 4-7). R13 is conventionally used as a stack pointer.

The non-user processor modes each have their own R13 and R14, and in 32-bit ARMs, PSR registers. FIQ mode additionally has its own R8-R12. When a mode change occurs because of interrupts, SWIs or traps, R14 of the new mode is set to a copy of R15, and in 32-bit ARMs the PSR of the new mode is copied from the PSR of the old mode. For further details of banked registers and mode changes, consult the appropriate ARM data sheet for the target processor.

Within an assembly language source, the current value of the program counter (PC) can be referred to as '.'. Usually, '.' is 8 bytes ahead of the instruction using it because of pipelining.

For example:

```
LDR R0,[.-8+offset]
```

loads a word at offset bytes from the current instruction. Please refer to the appropriate ARM data sheet for precise details.

4.2 Branch Instructions—B and BL

There are two branch instructions:

`B`	Branch
`BL`	Branch with Link

The syntax of these instructions is:

```
B{L}{condition} expression
```

where the expression evaluates to the branch destination address. If the address is within 32MB of the current program counter , it can be expressed directly as an offset.

On 32-bit address ARMs, branches of more than 32MB have to be effected by loading the destination address directly into the PC, or by adding a long offset to the PC using a value loaded into a register. Branch with Link saves the PC into R14 of the current bank.

4.2.1 Returning from a branch instruction

To return, use:

```
MOV    PC, R14
```

or if the link register has been saved on a stack, use

```
LDMFD  SP!, {...,PC}
```

Note: These instructions will not restore the original PSR.

The assembler automatically compensates for the effects of pipelining and prefetching when calculating offsets.

4.3 Data Processing Instructions

4.3.1 MOV and MVN

`MOV` (Move)	places an unchanged operand in the destination register
`MVN` (Move Negated)	places the bitwise inverse of the operand in the destination register

The syntax of these instructions is:

`opcode{condition}{S} destination,operand2`

`destination`	must be a register
`operand2`	may be any of:

- An 8-bit immediate constant, rotated right by a constant 0, 2, 4...30 bits:

  `#constant-expression{,constant-rotation}`

- A register shifted left, shifted logically right, shifted arithmetically right, or rotated right by a constant 0...31 bits:

  `register {,shift #constant-expression}`

  where shift is one of:

`LSL`	shift left
`LSR`	logical shift right
`ASR`	arithmetic shift right
`ROR`	rotate right

- A register, shifted as above by the amount given in another register:

  `register {,shift register}`

- A register rotated 1 bit right through the carry flag, (a 1 bit rotate right of a 33-bit value, after which the most significant bit of the register is the old value of the carry flag). The assembler syntax in this case is:

  `register, RRX`

For simple constants (for example #&FC000003) the assembler will generate the appropriate rotation for you.

4.3.2 Data processing instructions–arithmetic and logical

There are ten arithmetic and logical instructions:

`ADC`	Add with Carry: adds *`operand1`*, *`operand2`*, and the carry flag. Multi-word additions are made simple by this instruction.
`ADD`	Add: adds *`operand1`* to *`operand2`*.
`AND`	Bitwise AND: performs a bitwise AND between *`operand1`* and *`operand2`*.
`BIC`	Bit Clear: stores in the destination register the result of clearing from a copy of *`operand1`* any bit which is set in *`operand2`*.
`EOR`	Bitwise Exclusive OR: performs a bitwise Exclusive OR between *`operand1`* and *`operand2`*.
`ORR`	Bitwise OR: performs a bitwise OR between *`operand1`* and *`operand2`*.
`RSB`	Reverse Subtract: subtracts *`operand1`* from *`operand2`*.
`RSC`	Reverse Subtract with Carry: subtracts (*`operand1`* plus not carry) from *`operand2`*.
`SBC`	Subtract with Carry: subtracts (*`operand2`* plus not carry) from *`operand1`*.
`SUB`	Subtract: subtracts *`operand2`* from *`operand1`*.

The syntax of this group of instructions is:

```
opcode{condition}{S} destination,operand1,operand2
```

The *`destination`* and *`operand1`* must both be registers, and *`operand2`* should be as described for the `MOV` and `MVN` instructions (see ➲*4.3.1 MOV and MVN* on page 4-8).

With `ADD` and `ADC`, a carry is generated by 32-bit overflows; for subtractions it is generated if, and only if, underflow did not occur.

With `ADD`, `ADC`, `SUB`, `SBC`, `RSB` and `RSC` the V flag is set if signed overflow occurred. For example, when the carry into bit 31 was not equal to the carry out of that bit.

4.3.3 Data processing instructions–comparison operations

There are four comparison operations:

`CMN`	Compare Negated: adds the two operands, setting the condition flags according to the result.
`CMP`	Compare: subtracts `operand2` from `operand1`, setting all four condition flags according to the result.
`TEQ`	Test Equivalence: performs a bitwise Exclusive OR between the two operands, setting the Z flag if the result is zero.
`TST`	Test under Mask: performs a bitwise AND between the two operands, setting the Z flag if the result is zero.

The syntax of these instructions is:

```
opcode{condition}{P} operand1,operand2
```

The N and C flags may also be affected if a shift or rotation was involved in the construction of *`operand2`*.

Each of these instructions preserves its operands and produces no result other than updated PSR flags. *`operand1`* must be a register, and *`operand2`* must be as described for `MOV` and `MVN` (see ➲*4.3.1 MOV and MVN* on page 4-8).

If P is not specified, the PSR condition flags are set to the ALU condition flags after the operation (as described above), and the instructions behave as conventional status-setting comparisons.

With 26-bit ARMs, use of `P` allows direct manipulation of the PSR, as described below. Do not use `P` with 32-bit ARMs: instead use `MSR` and `MRS` (see ➲*4.4 PSR Transfer—MSR and MRS*).

26-bit modes

In 26-bit user mode, `{`*`opcode`*`}P` moves the result of the operation to the PSR, and sets the N, Z, C and V flags from the top four bits of the result.

In other 26-bit modes it sets the N, Z, C, V, I and F flags from the top six bits, and the mode bits from the bottom two bits of the result. A typical use of `{`*`opcode`*`}P` would be to change modes.

4.4 PSR Transfer—MSR and MRS

`MSR`	Move register to PSR: loads the processor status register.
`MRS`	Move PSR to register: stores PSR in a register.

The syntax of these instructions is:

```
MSR{condition} psrl,operand2
MRS{condition} destination,psrs
```

These instructions are available on 32-bit ARMs only. R15 cannot be used as the destination register. Please refer to your ARM data sheet for precise details.

psrl can be:

- *PSR{_fields}*

PS	is one of CPSR or SPSR
fields	consists of one or more of the letters `c x s f`:
f	indicates flags field (bits 31:24)
s	indicates status field (bits 23:16)
x	indicates extension field (bits 15:8)
c	indicates control field (bits 7:0)

Note that the field specifiers _ctl, _flg, and _all have been superseded by this new format.

psrs can be one of:

- `SPSR`
- `CPSR`

operand2 is as described in *4.3 Data Processing Instructions* on page 4-8.

User mode

In user mode the instructions behave as follows:

```
MSR CPSR_fc, op2          ; CPSR{N,Z,C,V} <- op2
MSR CPSR_f, op2           ; CPSR{N,Z,C,V} <- op2
MSR CPSR_c, op2           ; No effect
MRS Rd, CPSR              ; Rd <- CPSR{N,Z,C,V,I,F,M[4:0]}
MSR SPSR, op2             ; Not valid in user mode
MRS Rd, SPSR              ; Not valid in user mode
```

Privileged mode

In privileged modes the instructions behave as follows:

```
MSR CPSR_fc, op2          ; CPSR{N,Z,C,V,I,F,M[4:0]} <- op2
MSR CPSR_f, op2           ; CPSR{N,Z,C,V} <- op2
MSR CPSR_c, op2           ; CPSR{I,F,M[4.0]} <- op2
MRS Rd, CPSR              ; Rd <- CPSR{N,Z,C,V,I,F,M[4:0]}
MSR SPSR_fc, Rm           ; SPSR_mode{N,Z,C,V,I,F,M[4:0]} <- op2
MSR SPSR_f, Rm            ; SPSR_mode{N,Z,C,V} <- op2
MSR SPSR_c, Rm            ; SPSR_mode{I,F,M[4.0]} <- op2
MRS Rd, SPSR              ; Rd <- SPSR_mode{N,Z,C,V,I,F,M[4:0]}
```

4.5 Unsigned Word/Byte Data Transfer—LDR and STR

`LDR`	Load register from memory location.
`STR`	Store register to memory location.

These instructions come in two forms:

pre-indexed	The syntax of pre-indexed instructions is: `opcode{condition}{B} register,[base{,index}]{!}`
post-indexed	The syntax of post-indexed instructions is: `opcode{condition}{B}{T} register,[base]{,index}`

where:

`B`	specifies a byte instead of a word transfer (8 bits instead of 32)
`T`	is only allowed with a post-indexed load, and forces the transfer to take place as if in a non-privileged mode (from supervisor mode, for example)
`register`	is the destination of the load or source of the store
`base`	must be a register
`index`	specifies an offset from the base (see below)

For pre-indexed addressing, *`index`* is added to *`base`* to yield the load or store address. With post-indexed addressing, *`base`* gives the address for the load or store, and *`base+index`* is the value written back to *`base`*. In the pre-indexe case, ! enables writeback of *`base+index`* to *`base`*.

Index

`index` may be one of the following:

```
#{–}12-bit-constant-expression
{–}register {, shift #5-bit-constant-expression}
```

`shift` is explained in *4.3 Data Processing Instructions* on page 4-8. In this second form the value of *`index`* is the value in *`register`* shifted as specified.

`LDR` can also 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 *3.4.4 Organisational directives—END, ORG, LTORG and KEEP* on page 3-12).

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 *3.4.3 Describing the layout of store–^ and #* on page 3-12).

4.6 Halfword and Signed Data Transfer: LDRH, STRH, LDRSB, LDRSH

These instructions are available only on ARM processors which support version 4 (or later) of the ARM architecture, eg. ARM7TDMI. (The architecture version can be selected using the assembler options –`arch` and/or –`cpu`.

`LDRH`	Load unsigned halfword from memory location.
`STRH`	Store halfword to memory location.
`LDRSB`	Load signed byte from memory location.
`LDRSH`	Load signed halfword from memory location.

These instructions come in two forms:

pre-indexed — The syntax of pre-indexed instructions is:

```
opcode{condition}type register,[base{,index}]{!}
```

post-indexed — The syntax of post-indexed instructions is:

```
opcode{condition} type register,[base]{,index}
```

where:

`type`	is either H, SB, SH (with SB and SH only allowed for LDR), and specifies the width of the transfer (8 versus 16 bits) and whether sign extension takes place on the loaded word
`register`	is the destination of the load or source of the store
`base`	must be a register
`index`	specifies an offset from the base (see below)

For pre-indexed addressing, *`index`* is added to *`base`* to yield the load or store address. With post-indexed addressing, *`base`* gives the address for the load or store, and *`base+index`* is the value written back to *`base`*. In the pre-indexe case, ! enables writeback of *`base+index`* to *`base`*.

Index

`index` may be one of the following:

```
#{-}8-bit-constant-expression
{-}register
```

The form *`index`* can take is more restrictive than on the unsigned LDR and STR instructions. As a constant it must be 8-bit, and as a register it cannot have a shift applied.

Additionally these instructions can be used to transfer data to or from an address specified by a label (optionally with an offset), in the same manner as the unsigned LDR and STR instructions. See ➲*4.5 Unsigned Word/Byte Data Transfer—LDR and STR* on page 4-12 for details.

The form `LDRH` *`register,=expression`* (for example) is not supported.

4.7 Block Data Transfer—LDM and STM

`LDM`	Load multiple registers
`STM`	Store multiple registers

The syntax of these instructions is:

```
opcode{condition}type base{!},register-list{^}
```

The opcode is combined with one of eight instruction types with the mnemonics `DB`, `DA`, `IB`, `IA`, `FD`, `ED`, `FA`, and `EA` (note that the meaning of `FD`, `ED`, `FA` and `EA` varies according to whether a load or store is performed):

`STMDB`	Decrement *`base`* Before the store
`STMDA`	Decrement *`base`* After the store
`STMIB`	Increment *`base`* Before the store
`STMIA`	Increment *`base`* After the store
`LDMDB`	Decrement *`base`* Before the load
`LDMDA`	Decrement *`base`* After the load
`LDMIB`	Increment *`base`* Before the load
`LDMIA`	Increment *`base`* After the load
`STMFD`	Push registers to a Full stack, Descending (STMDB)
`STMED`	Push registers to an Empty stack, Descending (STMDA)
`STMFA`	Push registers to a Full stack, Ascending (STMIB)

`STMEA`	Push registers to an Empty stack, Ascending (STMIA)
`LDMFD`	Pop registers from a Full stack, Descending (LDMIA)
`LDMED`	Pop registers from an Empty stack, Descending (LDMIB)
`LDMFA`	Pop registers from a Full stack, Ascending (LDMDA)
`LDMEA`	Pop registers from an Empty stack, Ascending (LDMDB)
`base`	Contains the starting address for the transfer and can be any register except R15. If present ! requests writeback of the updated base address to base after the instruction is executed.
`register-list`	Is a comma-separated list of registers and/or register ranges enclosed in {}. A register range is two register names joined by a hyphen, and represents the registers named and all those between them. The directive RLIST (see ➲*3.4.10 Miscellaneous directives—ALIGN, NOFP, RLIST and ENTRY* on page 3-16) can also be used to create a list of registers.
`^`	In user mode, this sets the S bit to load the PSR along with the PC. In privileged modes, it forces transfer of the user mode registers.

A full stack is one in which the stack pointer points to the last data item written to it. An empty stack is one where the stack pointer points to the first free slot in it. A descending stack grows from high memory addresses to low, and an ascending stack vice-versa.

4.8 Multiply Instructions—MUL, MLA

`MUL`	32-bit result multiply ($32 \times 32 \rightarrow 32$)
`MLA`	32-bit result multiply and accumulate ($32 \times 32 + 32 \rightarrow 32$)

The syntax of these instructions is:

```
MUL{condition}{S} destination,operand1,operand2
MLA{condition}{S} destination,operand1,operand2,operand3
```

The destination and all operands must be registers. `MUL` multiplies *`operand1`* by *`operand2`*, and places the result in *`destination`*. `MLA` multiplies *`operand1`* by *`operand2`*, adds *`operand3`* to the product and places the result in *`destination`*. Both instructions work with signed and unsigned integers. For details of how to make multiply instructions execute quickly, see the Programming Techniques manual.

In all cases the result is truncated to 32 bits.

Certain combinations of operands should be avoided and are warned against by the assembler. The destination register should not be the same as *`operand1`* as this will give a meaningless result. R15 should not be used as the destination register, nor as an operand. See the appropriate ARM datasheet for further details.

4.9 Long Multiply Instructions—MULL, MLAL

`MUL`	32-bit result multiply ($32 \times 32 \rightarrow 32$)
`MLA`	32-bit result multiply and accumulate ($32 \times 32 + 32 \rightarrow 32$)

These instructions are available only on ARM processors which support version 3M (or later) of the ARM architecture: eg. ARM7DM, ARM7TDMI. (The architecture version can be selected using the assembler options `–arch` and/or `–cpu`.

The syntax of these instructions is:

```
UMULL{condition}{S} destLo,destHi,operand1,operand2
UMLAL{condition}{S} destLo,destHi,operand1,operand2
SMULL{condition}{S} destLo,destHi,operand1,operand2
SMLAL{condition}{S} destLo,destHi,operand1,operand2
```

The `UMUL` and `UMLA` instructions perform an unsigned multiply, whereas the `SMUL` and `SMLA` instructions treat both operands as two's complement signed numbers, and give a two's complement signed 64-bit result.

The destination and all operands must be registers. `MULL` multiplies *`operand1`* by *`operand2`*, and places the full 64-bit result in the pair of registers *`destLo`* and *`destHi`*. `MLAL` multiplies *`operand1`* by *`operand2`*, and adds the 64-bit result to the 64-bit value in *`destLo`* and *`DestHi`*.

Certain combinations of operands should be avoided and are warned against by the assembler. The registers *`destHi`*, *`destLo`*, and *`operand1`* must all be different. R15 should not be used as a destination register, nor as an operand. See the appropriate ARM datasheet for further details.

4.10 Single Data Swap—SWP

`SWP`	Single data swap

SWP swaps a byte or word quantity between a register and memory, locking the memory bus in the process to preserve atomic operation (where supported by external hardware). The syntax is:

```
SWP{condition}{B} destination,source,[base]
```

`Destination`, *`source`* and *`base`* must all be registers. B sets the width of the transfer to byte rather than word. The memory address is that in *`base`*; its contents are read, the *`source`* register is written to it, and the old memory contents are then stored in *`destination`*. The same register can serve as *`source`* and *`destination`*. R15 may not be used as the swap address, the *`source`* or the *`destination`*.

4.11 ARM to Thumb State Exchange—BX

Thumb: Entry into Thumb state is performed by the `BX` instruction. This has the syntax:

```
BX {condition} destination
```

where *`destination`* is a register holding a halfword-aligned branch destination address in bits 1 to 31. Bit 0 of *`destination`* determines the processor state:

Bit 0 = 0	Remain in ARM state
Bit 0 = 1	Enter Thumb state

4.12 Software Interrupt/Supervisor Call—SWI

`SWI`	software interrupt

This instruction is used by programs to communicate with the host operating system. The syntax is:

```
SWI constant-expression
```

The expression value is truncated to 24 bits (between &0 and &FFFFFF); it is ignored by the processor but is interpreted by operating system software.

4.13 Pseudo-Instructions—ADR and NOP

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

4.13.1 ADR

`ADR`	assemble address to register

Because the ARM has no “load effective address” instruction, the assembler provides `ADR`, which will always assemble 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.

4.13.2 NOP

`NOP`	no operation

This 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.

4.14 Generic Coprocessor Instructions

These are the generic coprocessor instructions implemented by all ARM processors with a coprocessor interface. Up to 16 coprocessors can be supported; all coprocessors have a number (CP#) in the range 0 to 15, and this must be specified in the instructions. Coprocessors 1 and 2 are conventionally floating point units. Coprocessor 14 is used as a debug channel on some processors, and coprocessor 15 is used for cache, write-buffer and memory management control in several ARM processors.

Coprocessors may have up to 16 directly addressable registers, C0-C15.

4.14.1 Coprocessor data transfers–LDC and STC

`LDC`	load data to coprocessor register from memory
`STC`	store data from coprocessor register to memory

These instructions transfer data between a coprocessor and memory. The syntax is:

```
op{condition}{L} CP#,Cd,[Rn {,#offset}]{!}
                 [Rn],#offset
                 program-or-register-relative-expression
```

The memory address can be expressed in one of three ways, as shown above. In the first, pre-indexed form, an ARM register, *Rn*, holds the base address to which an offset can be added if necessary. Writeback of the effective address to *Rn* can be enabled using !. The offset must be divisible by 4, and within the range -1020 to 1020 bytes. With the second, post-indexed form, write-back of *Rn*+*offset* to *Rn* after the transfer, is automatic. Alternatively, a *program-or-register-relative-expression* can be used, in which case the assembler will generate a PC- or register-relative, pre-indexed address; if it is out of range an error will result.

`L` appended to the instruction specifies a long transfer; otherwise a short transfer takes place. The meaning of this is coprocessor-specific.

4.14.2 Coprocessor data operations–CDP

`CDP`	coprocessor internal data processing operation

This instruction is used for internal coprocessor operations. The syntax is:

```
CDP{condition} CP#,CPOp,CRd,CRn,CRm{,CPOp2}
```

CPOp represents the coprocessor operation to be performed (four bits). Details of such operations are coprocessor-specific and can be found in the appropriate data sheet. The operation is performed on *CRn* and *CRm* and the result written to *CRd*. The second, optional, *CPOp2* field allows further variations on the operation (three bits).

4.14.3 Coprocessor register transfers–MCR and MRC

`MCR`	move data to coprocessor from ARM register
`MRC`	move data to ARM register from coprocessor

The syntax of these two instructions is:

```
op{condition} CP#,CPOp,Rd,Cn,Cm{,CPOp2}
```

`CPOp` is a 3-bit constant which specifies which variant of the instruction to perform. The selected operation is performed using the coprocessor registers *`Cn`* and *`Cm`*, and the result transferred to the ARM register *`Rd`*. If R15 is specified as the destination, only bits 28-31 of the result are transferred and are used to set the N, Z, C and V flags in the PSR without affecting the program counter. *`CPOp2`*, where present, is a 3-bit constant which sets the ARM condition flags, supporting the further coprocessor-specific sub-operations.

`MRC` is often used to read a coprocessor’s status register(s), while `MCR` is used to write its control register(s). `MRC`, with R15 as the destination, supports execution of ARM code conditional on the result of an earlier coprocessor operation (floating point compare, for example).

4.15 Floating Point Instructions

The ARM assembler supports a comprehensive floating point instruction set. Whether implemented by hardware coprocessor or software emulation, floating point operations are performed to the IEEE 754 standard. There are eight floating point registers, numbered F0 to F7. Floating point operations, like integer operations, are performed between registers.

Precision must be specified for many floating point operations where shown as *prec* below. The options are S (Single), D (Double), E (Extended) and P (Packed BCD). The format in which extended precision numbers are stored varies between FP implementations, and cannot be relied upon. The rounding mode, shown below as *round*, defaults to 'round to nearest', but can optionally be set in the appropriate instructions to: P (round to +infinity), M (round to –infinity) or Z (round to zero).

In all the following instruction patterns, *Rx* represents an ARM register, and *Fx* a floating point register.

4.15.1 Floating point data transfer–LDF and STF

LDF	load data to floating point register
STF	store data from floating point register

The syntax of these instructions is:

```
op{condition}prec Fd,[Rn,#offset]{!}
                     [Rn]{,#offset}
                     program-or-register-relative-expression
```

The memory address can be expressed in one of three ways, as shown above. In the first, pre-indexed form, an ARM register *Rn* holds the base address, to which an offset can be added if necessary. Writeback of the effective address to *Rn* can be enabled using !. The offset must be divisible by 4, and within the range -1020 to 1020 bytes. With the second, post-indexed form, writeback of Rn+*offset* to Rn after the transfer, is automatic. Alternatively, a program- or register-relative expression can be used, in which case the assembler will generate a PC- or register-relative, pre-indexed address; if it is out of range an error will result.

4.15.2 Floating point register transfer–FIX and FLT

FLT	integer to floating point	*Fn*:=*Rd*

The syntax of this instruction is:

```
FLT{condition}prec{round} Fn,Rd
                          Fn,#built-in-fp-constant
```

where *Rd* is an ARM register and *built-in-fp-constant* is one of 0, 1, 2, 3, 4, 5, 10 or 0.5.

FIX	floating point to integer	*Rd*:=*Fn*

The syntax of this instruction is:

```
FIX{condition}{round} Rd,Fn
```

4.15.3 Floating point register transfer–status and control

The following instructions transfer values between the FP coprocessor’s status and control registers, and an ARM general purpose register.

`WFS`	write floating point status	FPSR*:=Rd*
`RFS`	read floating point status	*Rd:*=FPSR
`WFC`	write floating point control	FPC:=*Rd* (privileged modes only)
`RFC`	read floating point control	*Rd:*=FPC (privileged modes only)

The syntax of the above four instructions is:

`opcode``{`*`condition`*`}` *`Rd`*

4.15.4 Floating point multiple data transfer–LFM and SFM

(Note that these instructions are not supported by some older versions of the Floating Point Emulator.)

`LFM`	load floating multiple
`SFM`	store floating multiple

These instructions are used for block data transfers between the floating point registers and memory. Values are transferred in an internal 96-bit format, with no loss of precision and with no possibility of an IEEE exception occurring, (unlike STFE which may fault on loading a trapping NaN). There are two forms, depending on whether the instruction is being used for stacking operations or not. The first, non-stacking, form is:

```
op{condition} Fd,count,[Rn]
                        [Rn,#offset]{!}
                        [Rn],#offset
```

The first register to transfer is *Fd*, and the number of registers to transfer is *`count`*. Up to four registers can be transferred, always in ascending order. The count wraps round at F7, so if F6 is specified with four registers to transfer, F6, F7, F0 and F1 will be transferred in that order. With pre-indexed addressing the destination/source register can be specified with or without an *`offset`* expressed in bytes; writeback of the effective address to *Rn* can be specified with !. With post-indexed addressing (the third form above) writeback is automatically enabled. Note that R15 cannot be used with writeback, and that offset must be divisible by 4 and in the range -1020 to 1020, as for other coprocessor loads and stores.

The second form adds a two-letter stacking mnemonic (below `ss`) to the instruction and optional condition codes. The mnemonic `FD` denotes a full, descending stack (pre-decrement push, post-increment pop), while `EA` denotes an empty, ascending stack (post-increment push, pre-decrement pop). The syntax is as follows:

```
op{condition}ss Fd,count,[Rn]{!}
```

`FD` and `EA` define pre- and post-indexing, and the up/down bit by reference to the form of stack required. Unlike the integer block-data transfer operations, only `FD` and `EA` stacks are supported. `!`, if present, enables writeback of the updated base address to *`Rn`*; R15 cannot be the base register if writeback is enabled.

The possible combinations of mnemonics are listed below:

`LFMFD`	load floating multiple from a full stack, descending (post-increment load)
`LFMEA`	load floating multiple from an empty stack, ascending (pre-decrement load)
`SFMFD`	store floating multiple to a full stack, descending (pre-decrement store)
`SFMEA`	store floating multiple to an empty stack, ascending (post-increment store)

4.15.5 Floating point comparisons–CMF and CNF

`CMF` `CMFE`	compare floating	compare *`Fn`* with *`Fm`*
`CNF` `CNFE`	compare negated floating	compare *`Fn`* with –*`Fm`*

The syntax of these instructions is:

`opcode``{`*`condition`*`}` *`Fn`*`,`*`Fm`*

`CMF` and `CNF` raise no exceptions and should be used to test for equality (Z clear/set) and unorderedness (V set/clear). To comply with IEEE 754, all other tests should use `CMFE` or `CNFE`, which may raise an exception if either of the operands is not a number.

4.15.6 Floating point binary operations

`ADF`	add	*Fd*:=*Fn*+*Fm*
`MUF`	multiply	*Fd*:=*Fn***Fm*
`SUF`	subtract	*Fd*:=*Fn*–*Fm*
`RSF`	reverse subtract	*Fd*:=*Fm*–*Fn*
`DVF`	divide	*Fd*:=*Fn*/*Fm*
`RDF`	reverse divide	*Fd*:=*Fm*/*Fn*
`POW`	power	*Fd*:=*Fn* to the power of *Fm*
`RPW`	reverse power	*Fd*:=*Fm* to the power of *Fn*
`RMF`	remainder	*Fd*:=remainder of *Fn*/*Fm*
`FML`	fast multiply	*Fd*:=*Fn***Fm*
`FDV`	fast divide	*Fd*:=*Fn*/*Fm*
`FRD`	fast reverse divide	*Fd*:=*Fm*/*Fn*
`POL`	polar angle	*Fd*:=polar angle of *Fn*,*Fm* (=ATN(*Fm*/*Fn*) whenever the quotient exists

The syntax of these instructions is:

binop{*condition*}*prec*{*round*} `Fd,Fn,Fm`

Fm can be either a floating point register, or one of the floating point constants #0, #1, #2, #3, #4, #5, #10 or #0.5. Fast operations produce results accurate to only single precision.

4.15.7 Floating point unary operations

`MVF`	move	*Fd*:=*Fm*
`MNF`	move negated	*Fd*:=–*Fm*
`ABS`	absolute value	*Fd*:=ABS(*Fm*)
`RND`	round to integral value	*Fd*:=integer value of *Fm* (using current rounding mode)
`URD`	unnormalised round:	*Fd*:= integer value of *Fm*, possibly in abnormal form
`NRM`	normalise	*Fd*:= normalised form of *Fm*
`SQT`	square root	*Fd*:=square root of *Fm*
`LOG`	logarithm to base 10	*Fd*:=log *Fm*
`LGN`	logarithm to base e	*Fd*:=ln *Fm*
`EXP`	exponent	*Fd*:=e^{Fm}
`SIN`	sine	*Fd*:=sine of *Fm*
`COS`	cosine	*Fd*:=cosine of *Fm*
`TAN`	tangent	*Fd*:=tangent of *Fm*
`ASN`	arc sine	*Fd*:=arc sine of *Fm*
`ACS`	arc cosine	*Fd*:=arc cosine of *Fm*
`ATN`	arc tangent	*Fd*:=arc tangent of *Fm*

The syntax of these instructions is:

```
unop{condition}prec{round} Fd,Fm
```

Fm can be either a floating point register or one of the floating point constants #0, #1, #2, #3, #4, #5, #10 or #0.5.

4.15.8 Floating point library

New applications which do not require compatibility with this instruction set should use the software floating point library instead. See *Chapter 16, Software Floating Point* for more details.

5 Thumb Instruction Set

This chapter describes the Thumb instruction set.

Thumb Instruction Set

5.1 Thumb Instruction Set—Overview

The Thumb instruction set is only available on variants of the ARM processor such as the ARM7TDM and ARM7TDMI. Please refer to the appropriate ARM data sheet for a full description of the Thumb programmer's model. This chapter provides the assembly language programmer with a reference guide to syntax and usage.

Thumb is a subset of the ARM instruction set: most ARM instructions are available to the Thumb programmer, though there are restrictions on the registers, operands and condition code bits that can be used.

Table 5-1: Thumb instruction set summary on page 5-3 lists the available instructions.

5.1.1 General restrictions

In the Thumb instruction set:

- Conditional execution may only be used on the branch (B) instruction.
- Only registers 0-7 (the *Lo registers*) may be accessed as general registers. Specific instructions may implicitly use registers 8-15 (the *Hi registers*): for example, the `PUSH` instruction implicitly accesses register 13—the stack pointer.
- There are no generic co-processor or floating point instructions: floating point operations are performed by a dedicated library which is provided as part of the C library in the Thumb tools release.
- Only two operands may be used on data processing instructions (except for `ADD` and `SUB` which can take three).
- The range of immediate fields is restricted: for example, the unconditional Branch instruction (B) has an 11-bit field.
- The only block data transfer instructions available are `LDMIA` and `STMIA` with write-back.

5.1.2 Option bits

Many of the option bits are not available in Thumb state. For example:

- There is no S bit in the Thumb data processing instructions, since most implicitly set the CPSR condition codes (for example, all the low-register operations).
- There is no writeback bit in the single data transfer instruction.

Mnemonic	Instruction	Lo register operand	Hi register operand	Condition codes set	See Section:
`ADC`	Add with Carry	✔		✔	5.3.1
`ADD`	Add	✔	✔	✔①	5.3.2-5.3.3, 5.3.5-5.3.6, 5.6.2
`AND`	AND	✔		✔	5.3.1
`ASR`	Arithmetic Shift Right	✔		✔	5.3.1, 5.3.4
`B`	Unconditional branch	✔			5.2.2
`B`*`xx`*	Conditional branch	✔			5.2.1
`BIC`	Bit Clear	✔		✔	5.3.1
`BL`	Branch and Link				5.2.3
`BX`	Branch and Exchange	✔	✔		5.7
`CMN`	Compare Negative	✔		✔	5.3.1
`CMP`	Compare	✔	✔	✔	5.3.1, 5.3.3, 5.3.5
`EOR`	EOR	✔		✔	5.3.1
`LDMIA`	Load multiple	✔			5.5
`LDR`	Load word	✔			5.4.1-5.4.4,
`LDRB`	Load byte	✔			5.4.1-5.4.2
`LDRH`	Load halfword	✔			5.4.1-5.4.2
`LSL`	Logical Shift Left	✔		✔	5.3.1, 5.3.4
`LDSB`	Load sign-extended byte	✔			5.4.1
`LDSH`	Load sign-extended halfword	✔			5.4.1
`LSR`	Logical Shift Right	✔		✔	5.3.1, 5.3.4
`MOV`	Move register	✔	✔	✔②	5.3.3, 5.3.5

Table 5-1: Thumb instruction set summary

Mnemonic	Instruction	Lo register operand	Hi register operand	Condition codes set	See Section:
`MUL`	Multiply	✔		✔	5.3.1
`MVN`	Move Negative register	✔		✔	5.3.1
`NEG`	Negate	✔		✔	5.3.1
`ORR`	OR	✔		✔	5.3.1
`POP`	Pop registers	✔			5.6.1
`PUSH`	Push registers	✔			5.6.1
`ROR`	Rotate Right	✔		✔	5.3.1
`SBC`	Subtract with Carry	✔		✔	5.3.1
`STMIA`	Store Multiple	✔			5.5
`STR`	Store word	✔			5.4.1-5.4.3
`STRB`	Store byte	✔			5.4.1-5.4.2
`STRH`	Store halfword	✔			5.4.1-5.4.2
`SWI`	Software Interrupt				5.8
`SUB`	Subtract	✔		✔	5.3.2-5.3.3, 5.6.2
`TST`	Test bits	✔		✔	5.3.1

Table 5-1: Thumb instruction set summary (Continued)

① The hi-register variant of `ADD` described in 5.3.5 and the `ADD SP` instruction described in 5.6.2 do not set the condition codes.

② The hi-register variant of `MOV` described in 5.3.5 does not set the condition codes.

5.2 Branch Instructions—B and BL

This section describes instructions for the following:

- Conditional branching
- Unconditional branching

5.2.1 Group 1: conditional branch

The operations in this group conditionally transfer execution control to the specified address. The syntax is:

```
Bcond      destination
```

where:

cond	is a condition code—see *Table 5-2: Conditional Branch codes* on page 5-6.
destination	is a program label within the range -512 to +508 of the current program counter.

The assembler also allows the following, which branches if the specified condition is *not* true:

```
BNcond     destination
```

The conditions are shown in *Table 5-2: Conditional Branch codes* on page 5-6.

5.2.2 Group 2: unconditional branch

This operation unconditionally transfers execution control to the specified address.

The syntax is:

```
B          destination
```

where *destination* is a program label within the range -2048 to +2044 of the current program counter.

5.2.3 Group 3: Branch with Link

This operation writes the return address into the link register and then unconditionally transfers execution control to the specified address. The syntax is:

```
BL         destination
```

where *destination* may be:

- a program label
- an external label +/- an immediate offset

HS (Higher or Same) and LO (Lower than) are synonyms for CS and CC respectively.

B: branch	BN: branch if not	if the condition codes are as follows:
`EQ`	`NE`	Z set
`NE`	`EQ`	Z clear
`CS`	`CC`	C set
`CC`	`CS`	C clear
`MI`	`PL`	N set
`PL`	`MI`	N clear
`VS`	`VC`	V set
`VC`	`VS`	V clear
`HI`	`LS`	C set and Z clear
`LS`	`HI`	C clear or Z set
`GE`	`LT`	N set and V set, or N clear and V clear
`LT`	`GE`	N set and V clear, or N clear and V set
`GT`	`LE`	Z clear, and either N set and V set or N clear and V clear
`LE`	`GT`	Z set, or N set and V clear, or N clear and V set

Table 5-2: Conditional Branch codes

5.2.4 Effect on Condition Codes

The branch operations do not affect the condition codes.

5.3 Data-processing Instructions

This section describes the following:

- two-operand address format
- three-operand address format
- immediate operations
- immediate shifts
- hi-register operations
- effective address calculation
- condition codes

5.3.1 Group 1: two-operand format

This group of instructions has the syntax:

```
opcode    dest/source1, source2
```

where *dest/source1* and *source2* are all registers in the range 0 to 7.

If *dest* and *source1* are the same register and the operation is not two-operand-only (ie. is not `CMP`, `CMN`, `NEG` or `MVN`), the assembler also allows:

```
opcode    dest, source1, source2
```

opcode may be one of:

`AND`	*source1* & *source2* -> *dest*
`EOR`	*source1* ^ *source2* -> *dest*
`LSL`	*source1* << *source2* -> *dest*
`LSR`	*source1* >> *source2* -> *dest*
`ASR`	*source1* >> *source2* -> *dest* (arithmetic or sign preserving)
`ADC`	*source1* + *source2* + C -> *dest*
`SBC`	*source1* - *source2* - !C -> *dest*
`ROR`	*source1* ROR *source2* -> *dest* (rotate right)
`TST`	set condition codes only on *source1* & *source2*
`NEG`	-*source2* -> *dest*
`CMP`	set condition codes only on *source1* - *source2*
`CMN`	set condition codes only on *source1* + *source2*
`ORR`	*source1* | *source2* -> *dest*
`MUL`	*source1* * *source2* -> *dest*
`BIC`	*source1* & ~*source2* -> *dest*
`MVN`	~*source2* -> *dest*

5.3.2 Group 2: three-operand format

This group has the syntax:

```
opcode    dest, source1, source2
```

where:

`dest` and `source1`	are registers in the range 0 to 7
`source2`	is either a register in the range 0 to 7 or an immediate 3-bit constant in the range 0 to 7

The assembler also allows the following form if *dest* and *source1* are the same register:

```
opcode    dest/source1, source2
```

where *opcode* may be one of:

```
ADD       source1 + source2 -> dest
SUB       source1 - source2 -> dest
```

5.3.3 Group 3: immediate operations

This group has the syntax:

```
opcode    dest/source, #N
```

where:

`dest/source`	is a register in the range 0 to 7
`N`	is an 8-bit constant in the range 0 to 255

opcode may be one of:

```
MOV       #N -> dest
CMP       set condition codes only on  source - #N
ADD       source + #N -> dest
SUB       source - #N -> dest
```

5.3.4 Group 4: immediate shifts

This group has the syntax:

```
opcode    dest, source, #N
```

where:

dest	is a register in the range 0 to 7
source	is a register in the range 0 to 7
N	is a 5-bit constant in the range 0 to 31

The assembler also allows the following form if *dest* and *source* are the same register.

```
opcode    dest/source, #N
```

opcode may be one of:

```
LSL       source << N -> dest
LSR       source >> N -> dest
ASR       source >> N -> dest (arithmetic or sign preserving)
```

5.3.5 Group 5: hi register operations

This group has the syntax:

```
opcode    dest/source1, source2
```

where *dest* and *source1* are registers in the range 0 to 15. At least one of *dest* or *source1* must be in the range 8 to 15, otherwise the instruction's behaviour is undefined.

The assembler also allows the following format for the `ADD` instruction, if *dest* and *source1* are the same register:

```
opcode    dest, source1, source2
```

opcode may be one of:

```
ADD       source1 + source2 -> dest
CMP       set condition codes only on source1 - source2
MOV       source2 -> dest
```

5.3.6 Group 6: effective address calculation

This group has the syntax:

```
ADD         dest, source, #N
```

where:

dest	is a register in the range 0 to 7
source	is either register 13 or 15. If *source* is R15, the assembler also allows the format: `ADR    dest, program-label`
N	is a word-aligned constant in the range 0 to 1024 (bits 1:0 of *N* must be 0)

5.3.7 Effect on condition codes

The Thumb data-processing instructions affect the condition codes in the following ways:

- All operations in Groups 1 to 4 and the `CMP` instruction in Group 5 affect the Z and N flags on the result of the operation.
- `ADC` sets the C flag if the result overflows, otherwise the C flag is cleared.
- `SBC` clears the C flag if the result overflows, otherwise the C flag is set.
- `ADC` and `SBC` set the V flag if a signed overflow occurs (ie. bit 31 does not contain the correct sign of the result), otherwise it is cleared.
- The shift operations (`LSL`, `LSR`, `ASR`, `ROR`) set the C flag to the last bit shifted out. The setting of the V flag is not defined after a shift operation.
- `NEG` sets the flags as though a `SUB` *`dest`*`, #0,`*`source2`* operation were performed.
- Bitwise operations (`AND`, `EOR`, `ORR`, `BIC`, `MOV`, `MVN` and `TST`) do not affect the C or V flags.

5.4 Single Data Transfer Instructions—LDR and STR

This section describes the following:

- single data transfer with base + register offset addressing
- single data transfer with base + immediate offset addressing
- single data transfer with stack pointer + immediate offset addressing
- single data transfer with program counter + immediate offset addressing

5.4.1 Group 1: load/store with register offset

This group has the following syntax:

```
opcode    source/dest, [base, offset]
```

source/dest, *base* and *offset* are all registers in the range 0 to 7.

opcode may be one of :

`LDR`	load a 32-bit value from *base* + *offset*
`STR`	store a 32-bit value at *base* + *offset*
`LDRH`	load a 0-extended 16-bit value from *base* + *offset*
`LDRSH`	load a sign-extended 16-bit value from *base* + *offset*
`STRH`	store a 16-bit value at *base* + *offset*
`LDRB`	load a 0-extended 8-bit value from *base* + *offset*
`LDRSB`	load a sign-extended 16-bit value from *base* + *offset*
`STRB`	store an 8-bit value at *base* + *offset*

Note: `LDSH` and `LDSB` may be used as alternative opcodes for `LDRSH` and `LDRSB`.

5.4.2 Group 2: load/store with immediate offset

This group has the syntax:

opcode *source/dest*, [*base*, #*N*]

where:

source/dest	is a register in the range 0 to 7
base	is a register in the range 0 to 7.
N	is a 5-bit constant—the range of this constant depends on the size of the object being transferred. See *Table 5-3: Offset ranges for base + immediate addressing*, below.

Data type	Offset range	Alignment
Byte	0 to 31	Any
Halfword	0 to 62	Must be multiple of 2
Word	0 to 124	Must be a multiple of 4

Table 5-3: Offset ranges for base + immediate addressing

The assembler also accepts the following form if *N* is zero:

opcode *source/dest*, [*base*]

opcode may be one of:

LDR	load a 32-bit value from *base* + *N*
STR	store a 32-bit value at *base* + *N*
LDRH	load a 16-bit value from *base* + *N*
STRH	store a 16-bit value at *base* + *N*
LDRB	load an 8-bit value from *base* + *N*
STRB	store an 8-bit value at *base* + *N*

5.4.3 Group 3: SP-relative load/store

This group has the syntax:

opcode *source/dest*, [SP, #*N*]

where:

source	is a register in the range 0 to 7
dest	is a register in the range 0 to 7
N	is a word-aligned 8-bit constant in the range 0 to 1024 (bits 1:0 of N must be 0)

The assembler also accepts the following form if `N` is zero:

```
opcode    source/dest, [SP]
```

`opcode` may be one of:

`LDR`	load a 32-bit value from `SP + N`
`STR`	store a 32-bit value at `SP + N`

5.4.4 Group 4: PC-relative load

This group has the syntax:

```
LDR       source, [PC, #N]
LDR       source, [PC]
```

where:

source	is a register in the range 0 to 7.
N	is an 8-bit constant in the range 0 to 1024 with an alignment of 4 (ie. it must be a multiple of 4).

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.

5.4.5 Effect on condition codes

The single data transfer operations do not affect the condition codes.

5.5 Block Data Transfer Instructions—LDMIA and STMIA

The block data transfer instructions have the format:

```
    opcode    base!, {register-list}
```

where:

base	is a register in the range 0 to 7. Note that write-back is enforced.
register-list	is a comma-separated list of registers and/or register ranges. A register range is two register names joined by a hyphen. All registers must be in the range 0 to 7.

opcode may be one of:

`LDMIA`	load the block of registers specified in register-list
`STMIA`	store the block of register specified in register-list

Transfer starts at the address specified by *base*, beginning with the lowest-numbered register in *register-list*.

The *base* register is updated to point to the memory location immediately following the address to which the last memory word was transferred.

If the *base* register is also specified in the *register-list* of an `LDMIA` instruction, the final value of *base* is the value loaded from memory, and not the updated pointer.

If the *base* register is also specified in the *register-list* of an `STMIA` instruction, the value stored depends on whether *base* is the lowest-numbered register in *register-list*. If so, the unmodified value is stored, otherwise the updated value is stored (ie. the value of *base* when the instruction completed).

5.5.1 Effect on condition codes

The block data transfer operations do not affect the condition codes.

5.6 Stack Operations—PUSH and POP

This section describes

- Pushing and popping registers
- Stack pointer adjustment operation

5.6.1 Group 1: PUSH/POP instructions

This group has the syntax:

```
opcode    {register-list}
```

where *`register-list`* is a comma-separated list of registers or register ranges. A register range is two register names joined by a hyphen. All registers must be in the range 0 to 7 plus R14 for `PUSH` or R15 for `POP`.

`opcode` may be one of:

`PUSH`	Push a block of registers, optionally including the link register, from the implicit program stack.
`POP`	Pop a block of registers, optionally including the program counter, from the implicit program stack.

The implicit program stack is a full descending stack using R13 as the stack pointer.

5.6.2 Group 2: stack pointer adjust

This group has the syntax:

```
opcode    R13, #N
```

where:

`N`	is an immediate constant in the range -508 to +508.

The assembler also allows the form:

```
opcode    R13, R13, #N
```

`opcode` may be one of :

```
ADD       R13 + N -> R13
SUB       R13 - N -> R13
```

5.6.3 Effect on condition codes

The stack operations do not affect the condition codes.

5.7 Thumb to ARM State Exchange—BX

Switches from Thumb to ARM state are performed using the `BX` instruction. This has the syntax:

```
BX      dest
```

where *`dest`* is a register in the range 0 to 15.

This instruction transfers control to the address contained in bits 1 to 31 of *`dest`*, in the processor state determined by bit 0:

Bit 0 = 0	enters ARM state
Bit 0 = 1	remains in Thumb state

Thumb code symbols automatically have bit 0 set when they are declared. The `BX` instruction uses this fact to allow transparent state changes. The caller does not need to know what state the callee executes in, as this information is contained in the destination label. Provided the link register is correctly set up and the called routine returns with a `BX LR` instruction, the called routine will return to the correct execution state on completion.

5.7.1 Effect on condition codes

The `BX` instruction does not affect the condition codes.

5.8 Software Interrupt

This operation performs a software interrupt. The return address is written into the link register. The processor is then switched to ARM state and control is transferred to location 8 (the SWI vector) in supervisor (SVC) mode.

The syntax is:

```
SWI       SWI-No
```

where *`SWI-No`* is a software interrupt number in the range 0 to 255.

5.8.1 Effect on condition codes

The `SWI` instruction does not affect the condition codes.

5.9 Pseudo Instructions — MOV and NOP

The Assembler supports several pseudo instructions which are translated into the appropriate combination of Thumb instructions at assembly time.

5.9.1 ADR

```
ADR reg, label
```

places address of `label` in `reg`.

`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.

5.9.2 MOV

```
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.

5.9.3 NOP

```
NOP
```

The Thumb `NOP` pseudo instruction generates a `MOV R8,R8` instruction. (The ARM `NOP` generates a `MOV R0, R0` instruction. Hence the condition codes are unaltered by ARM or Thumb `NOP`s.

6 Linker

This chapter introduces the ARM linker.

6.1 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.

The ARM linker, armlink, accepts as input:

- one or more separately compiled or assembled object files written in ARM Object Format (see *6.7 ARM Object Format* on page 6-15)
- optionally, one or more object libraries in ARM Object Library Format (see *6.7 ARM Object Format* on page 6-15)

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 *6.4 Area Placement and Sorting Rules* on page 6-11 for details)
- relocates (possibly partially) relocatable values
- generates an output image, possibly comprising several files (or a partially linked object file instead)

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

- ARM Object Format (see *6.7 ARM Object Format* on page 6-15 for a synopsis, and *Chapter 10, ARM Object Format Decoder* for full details);
- Plain binary format, relocated to a fixed address (see *6.8 Plain Binary Format* on page 6-16 for details)
- ARM Image Format (see *6.9 ARM Image Format* on page 6-16 for a synopsis)
- VLSI-extended Intellec Hex Format, suitable for driving the Compass integrated circuit design tools (see *6.10 Extended Intellec Hex Format (IHF)* on page 6-18 for details)
- ARM Shared Library Format: a read-only position-independent re-entrant shareable code segment (or shared library), written as a plain binary file, together with a stub containing read-write data, entry veneers, etc., written in ARM Object Format. (See *6.11 ARM Shared Library Format* on page 6-18 for details)
- ARM Overlay Format: a root segment written in ARM Image Format, together with a collection of overlay segments, each written as a plain binary file. (See *6.12 Overlays* on page 6-28 for details). A system of overlays may be static (each segment bound to a fixed address at link time), or dynamic (each segment may be relocated loading)
- Scatter-loading format: this 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 the regions to be loaded into memory at addresses different to their execution addresses. (See *6.14 Scatter Loading* on page 6-41)

6.2 Using the Linker

The format of the link command is:

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

If present, `input–file–list` is a list of one or more object files and libraries; this is described in ➲*6.2.5 Input file list* on page 6-9. Input files, libraries and linker options may also be given in a file used as an argument to a -VIA option (see ➲*6.2.1 General options* for details of the -VIA option). This is especially convenient when the input file list is long.

If an option keyword takes an argument, a space must separate the argument from the keyword.

Option keywords are case insensitive. In the remainder of this section the abbreviations recognised by armlink are shown capitalised.

6.2.1 General options

`–Help`

Print a screen of help text summarising the linker's options and exit with a good return code.

`–Output` *`name`*

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

`–NoDebug`

Turn off the inclusion of debug tables in the output file. armlink includes debug tables in the output file by default.

If objects are compiled without debugging enabled, the linker will still include low-level symbolic debugging data unless the `–NoDebug` option is specified.

`–Info <topic>`

Print the information about a number of specified topics during the link process. `<topics>` is a comma separated list of keywords.

A keyword may be one of the following:

`Totals`	Report the total code and data sizes in the image. The totals are broken down into separate totals for object files and library files.
`Sizes`	Give a more detailed breakdown of the code and data sizes on an object by object basis.
`Interwork`	List all calls for which the ARM/Thumb interworking veneer was necessary.
`Unused`	List all unused AREAs when used in conjuntion with the `–Remove` option.

`–LIST` *`file`*

Re-direct the standard output stream to file. This is especially useful in conjunction with `-MAP`, `–Xref` and `–Symbols`.

`–ERRORS` *`file`*

Re-direct 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.

`–VIA` *`file`*

Read a further list of input file names 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 file name and option is given on a separate line. There may be multiple -VIA options, and `–VIA` options may be nested.

`–Verbose`

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

`–MAP`

Create 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 re-directed to a file using the host's stream re-direction facilities or the `–LIST` option).

`-Xref`

List 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 re-directed to a file using the host's stream re-direction facilities or the `-LIST` option).

`-Symbols` *`file`*

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

6.2.2 Output format options

The following options each select a different output format (and, hence, are mutually exclusive):

`-AIF`

Generate an output image in executable ARM Image Format (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 *6.2.4 Special options* on page 6-7). AIF is described in section *6.9 ARM Image Format* on page 6-16.

`-AIF -Relocatable`

Generate a relocatable AIF image which when entered self-relocates to its load address.

`-AIF -Relocatable -Workspace` *`n`*

Generate 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 ➲*6.2.4 Special options* on page 6-7).

Some fields of the AIF header and the self-relocation code generated by the linker can be customised 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 ➲*6.9 ARM Image Format* on page 6-16).

`-AOF`

Generate partially linked output, in ARM Object Format (AOF), suitable for inclusion in a subsequent link step. AOF is described in ➲*6.7 ARM Object Format* on page 6-15, and in ➲*Chapter 10, ARM Object Format Decoder*.

`-BIN`

Generate 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 ➲*6.2.4 Special options* on page 6-7). Plain binary images are described in ➲*6.8 Plain Binary Format* on page 6-16.

`-BIN -AIF`

Generate 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 ➲*6.2.4 Special options* on page 6-7). 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 ➲*6.2.4 Special options* on page 6-7); otherwise, by default, data are linked immediately following code. This option directly supports images with code in ROM and data in RAM.

`-IHF`

Generate a plain binary image encoded in extended Intellec Hex Format. The output is ASCII-coded.

`-SHL` *`file`*

Generate 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 sharable library image. See ➲*6.11.6 Describing a shared library to the linker* on page 6-26 for further details.

`-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 reentrant client. Usually, a client application is not reentrant (multi-threadable) so the default non-reentrant stub is more often useful.

`-SPLIT`

This option 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 ➲*6.2.4 Special options* on page 6-7). The separate output files are named as in ➲*Table 6-1: Linker output filenames* on page 6-6.

`-OVerlay` *`file`*

Generate 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. See ➲*6.12 Overlays* on page 6-28 for details.

	Output file names	
Linker command-line options	**read-only**	**read-write**
-o file -SPLIT -RO robase -RW rwbase	file.ro	file.rw
-o file -SPLIT -B robase -DATA rwbase	file	file.dat

Table 6-1: Linker output filenames

`-OVerlay` *`file`* `-Relocatable`

Generate 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. See ➲*6.12 Overlays* on page 6-28 for details.

6.2.3 Scatter loading command-line options

The linker generates a scatter loaded image when the option `-SCATTER` *`file`* is present on the linker command line. Several options are ignored when `-SCATTER` is present. These options are:

`-RO-base` and `-Base`

`-RW-base` and `-DATA`

`-SPLIT`

`-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.

When `-BIN` is present on the command line, the output file specification will be treated as a directory name. Each load region will be placed in a separate file in that subdirectory. The file name will be the load region name. Hence load region names must not contain characters unacceptable to the file system.

Specifying `-AIF` or `-AIF -BIN` will generate 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 `-AIF -BIN`. A linker warning is generated if `-AIF` is supplied without `-BIN`.

These options will produce a directory named *xxxx* containing binary files:

```
-SCATTER file -BIN -o xxxx
```

These options will produce a single file named *yyyy* containing an AIF header and the load regions:

```
-SCATTER file -AIF -BIN -o yyyy
```

6.2.4 Special 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.

`-RO-base` *`base-address`*
`-Base` *`base-address`*

Set 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. Currently, this option is only meaningful if the output type is `-BIN -AIF`, `-BIN -SPLIT` or `-IHF` *`-SPLIT`*.

`-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` and that letter 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 6-9).

`-Case`

Make the matching of symbol names case insensitive.

`-MATCH` *`flags`*

Set the last-gasp symbol matching options and the pc-relative implies code relocation default. Each option is controlled by a single bit in flags, as follows:

0x01:	match an undefined symbol of the form *`_sym`* to a symbol definition of the form *`sym`*
0x02:	match an undefined symbol of the form *`sym`* to a symbol definition of the form *`_sym`*
0x04:	match an undefined symbol of the form *`Module_Symbol`* to a definition of the form *`Module.Symbol`*
0x08:	match an undefined symbol of the form *`symbol__type`* to a definition of the form *`symbol`*
0x10:	treat 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). Take care not to override options accidentally when using `–MATCH` from the command line.

`-FIRST` *`object(area)`*
`-LAST` *`object(area)`*

Place the area named *`area`* from the *`object`* named object first or last, in the output.These options 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.

`-Remove`

Remove unused areas from the output. An area is used if:

- it is the area containing the entry point
- it is referred to from a used area.

`-DUPOK`

Allow duplicate symbols (a warning is displayed). An area can be included more than once if this option is used. The 2nd, 3rd, 4th etc. copies of the area will not be included in the image providing unused area elimination is not disabled (see above).

`-Unresolved` *`symbol`*

Match each reference to an undefined symbol to the global definition of *`symbol`*. Note that *`symbol`* must be both defined and global, otherwise it too 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.

`-NOZEROpad`

When generating a plain binary image, the linker expands zero-initialised areas with zero bytes in the image by default. This is so that the area will be zero-initialised when the image is loaded directly into memory. At runtime, the `-NOZEROpad` option sets memory between `Image$$Z1$$Base` and `Image$$Z1$$Limit` to zero. The ARM C Library does this.

6.2.5 Input file list

On the command line, the input file list is one or more file names separated by spaces, together with the files listed in arguments to `-VIA` options; see ➲*6.2.1 General options* on page 6-3. 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 in `-VIA` file `vf1`, etc.

Each of the files in the input list must be in ARM Object Format (compiled or assembled files) or ARM Object Library Format (libraries).

6.3 Library Module Inclusion

An object file may contain references to external objects (functions and variables), which 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, some object file or some already-included library module makes a *non-weak* reference to it

The linker processes its input list as follows:

- First, 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: references which are allowed to remain unsatisfied, and which do not cause a library member to be loaded.

Then, in the order that they appear in the input file list, the libraries are processed as follows:

- the library is searched for members containing symbol definitions which match currently unsatisfied, non-weak references
- each such member is loaded, satisfying some unsatisfied references (including possibly *weak* ones), and maybe, creating new unsatisfied references (again, maybe including weak ones)
- 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 round brackets after the library name. Note that there should be no space between the library name and the opening bracket. Multiple module names must be separated by a comma. There must be no space in the list of module names.

6.4 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 except AOF, except where overridden by a `-FIRST` or `-LAST` option, the linker sorts the set of areas first by attribute, then by area name.

The read-only parts of the image are then collected into one contiguous region which can be protected at run time 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 ALIGN=*n* attribute of the ARM assembler AREA directive.

Portions of the image associated with a particular language run-time 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 initialised data
zero-initialised (uninitialised) data
debugging tables

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

Debugging tables are included only if the linker's `-Debug` option is used. 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 unordered 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.

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.

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, then 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 $2^{(\text{area alignment})}$ (but most commonly, area alignment is 2, requiring only word alignment).

6.5 Linker Pre-Defined 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 *6.14.8 Initialisation* on page 6-46 for details of the symbols generated when using 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$$RW$$Base`	Address of the start of the read/write area (usually contains data)
`Image$$RW$$Limit`	Address of the byte beyond the end of the read/write area
`Image$$ZI$$Base`	Address of the start of the 0-initialised area (zeroed at image load or start-up time)
`Image$$ZI$$Limit`	Address of the byte beyond the end of the zero-initialised area

Object/area-related symbols

`areaname$$Base`	The address of the start of the consolidated area called `areaname`
`areaname$$Limit`	The address of the byte beyond the end of the consolidated area called `areaname`

`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` 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

It is poor programming practise to rely on `Image$$RO$$Limit` being the same as `Image$$RW$$Base`.

Note: 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 run-time 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.

6.6 The Handling of Relocation Directives

The linker implements the relocation directives defined by ARM Object Format. In this section you will read about their function.

6.6.1 The subject field

A relocation directive describes the relocation of a single subject field, the value of 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. The relocation of sequences of instructions is discussed later in this section.

6.6.2 The relocation value

A relocation directive either refers 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.

6.6.3 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.

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

A second special case of PC-relative relocation (specified by REL_B being set in the rel_flags field—see ➲*6.7 ARM Object Format* on page 6-15 for details) applies when the relocation value is the value of a code symbol. 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 optimisation to be performed on reentrant code. For more information about tail call continuation, please see ➲*19.5 Function Entry* on page 19-16.

6.6.5 Additive relocation

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

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

6.6.6 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 reentrant 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
```

6.6.7 The relocation of instruction sequences

The linker recognises the following instruction sequences as defining 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 recognised may be further restricted to 1, 2 or 3 instructions only by the relocation directive itself. For more information, see ➲*6.6 The Handling of Relocation Directives* on page 6-13.

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 non-zero, 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, then 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.

6.7 ARM Object Format

An object file written in ARM Object Format (AOF) consists of any number of named, attributed *areas*. Attributes include: read-only, reentrant, code, data, position independent etc.—for details see ➲*21.2.11 Attributes + alignment* on page 21-16). Typically, a compiled AOF file contains a read-only code area, and a read-write data area (a zero-initialised data area is also common, and re-entrant code uses a separate based area for address constants).

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.

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
- re-writes symbol-relative relocation directives between merged areas, as area-based relocation directives belonging to the merged area
- minimises the symbol table

Unresolved references remain unresolved, and the output AOF file may be used as the input to a further link step.

6.8 Plain Binary Format

An image in plain binary format is a sequence of bytes to be loaded into memory at a known address. How this address is communicated to the loader, and where to enter the loaded image, are not the business of the linker.

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)
- complete performance of all relocation directives

Input areas having the read-only attribute are placed at the low-address end of the image; initialised writeable areas follow; zero initialised areas are consolidated at the end of the file where a block of zeroes of the appropriate size is written.

Note:if the binary file is part of a scatter loaded application, the zero initialised areas are not present. The initialisation information generated by the linker enables the application initialisation to generate the zero initialised areas at run time.

6.9 ARM Image Format

At its simplest, a file in ARM Image Format (AIF) is a plain binary image preceded by a small (128 byte) header which describes what follows. 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 to which you have access which 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 whatever address the image has been loaded at; 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-initialised area.
- Finally, the enwrapped image is entered at the (unique) entry point found by the linker in the set of input object modules.

AIF flavours

Three flavours of AIF are supported:

Executable AIF	can be loaded at its load address and entered at the same point (at the first word of the AIF header). It prepares itself for execution by relocating itself, zeroing its own zero-initialised 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.
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	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.

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).

AIF header

The AIF header is specified fully in *21.1.2 The Layout of AIF* on page 21-4.

The contents of some fields of the AIF header (such as the program exit instruction) can be customised by providing a template for the header in an area with the name AIF_HDR and a length of 128 bytes, in the first object module in the list of object modules given to armlink.

Similarly, the self-move and self-relocation code appended by the linker to a relocatable AIF image can customised by providing an area with the name AIF_RELOC, also in the first object module in the input list.

6.10 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.

6.11 ARM Shared Library Format

ARM Shared Library format directly supports:

- shared code in ROM
- single-address-space, loadable, shared libraries.

Output in ARM Shared Library Format generates 2 files

- a read-only, position-independent, reentrant shared library, written as a plain binary file
- a *stub* file containing read-write data, entry vectors, etc., written in ARM Object Format, with which clients of the shared library will subsequently be linked

Optionally, a shared library can contain a read-only copy of its initialised static data which can be copied at run time to a zero-initialised place holder in the stub. Such data must be free of relocation directives.

The outputs are created from:

- a set of input object files, between/from which there must be no unresolved symbolic references
- a description of the shared library which includes the list of symbols to be exported from the library to its clients and a description of the data areas to be initialised at run time by copying from the shared library image

Code to be placed in a shared library must be compiled with the reentrant option, or if assembled, must conform to the shared library addressing architecture described in ➲*6.11.2 The shared library addressing architecture* on page 6-20.

6.11.1 Stub properties

The linker can generate a non-reentrant stub for use by non-reentrant client applications, or a reentrant stub which can be linked into another shared library or reentrant application.

The details of how a stub is initialised at load time or run time (so that a call to a stub entry point becomes a call to the corresponding library function) are system-specific. The linker provides a general mechanism for attaching code and data to both the library and the stub to support this.

In particular:

- The linker appends a table of offsets of library entry points (the Exported Function Table or EFT) to the library, followed by a parameter block specified in the shared library description input to the linker.
- The linker writes the same parameter block to the stub, and initialises the stub entry vector so that the first call through any element of it is to the dynamic linking code. The dynamic linking code can patch the stub entry vector given only a pointer to its shared library's EFT. After dynamic linking, execution resumes by calling through the stub vector entry which initially invoked the dynamic linking code. The dynamic linking code will not be called again (for this shared library).
- If the library contains a read-only copy of its initialised static data, the linker writes the length and relocatable address of the place holder immediately before the stub parameter block, and writes the length and offset of the initialising data immediately before the library parameter block. For uniformity of dynamic linking, the length and address or offset can be zero, denoting that neither initialising data nor a stub place holder are present in this shared library (though they may be present in other shared libraries handled by the same dynamic linker).

Provided the stub entry vector is writeable, the only system-specific part of the matching of the stub to (a compatible version of) its library is the location of the library itself. In general, this is expected to be a system service, though it would be equally possible to search a table at a fixed address, or simply to search the whole of ROM for a named library (the linker provides support for prepending a name, the offset of the EFT, and anything else that can be assembled to a shared library).

Alternatively, in support of more protected systems, the patching code can simply be a call to a system service which locates the matching library and patches the entry vector.

The patching of shared library entry vectors by the loader at load time is not directly supported. However, it would be a relatively simple extension to AIF to support this. In general, it is considered more efficient to patch on demand in systems with multiple shared libraries.

The user-specified parameter block mechanism allows fine control over, and diagnosis of the compatibility of a stub with a version of its shared library. This supports a variety of approaches to foreverness, without mandating foreverness where it would be inappropriate. This issue is discussed in ➲*6.11.5 Versions, compatibility and foreverness* on page 6-24.

6.11.2 The shared library addressing architecture

The central issue for shared objects is that of addressing their clients' static data.

On ARM processors, it is very difficult, and/or inefficient, to avoid the use of *address constants* when addressing static data, particularly the static data of separately compiled or assembled objects; (an address constant is a pointer which has its value bound at link time—in effect, it is an execution-time constant).

Typically, in non-reentrant code, these address constants are embedded in each separately compiled or assembled code segment, and are, in turn, addressed relative to the program counter. In this organisation, all threadings of the code address the same, link-time bound static data.

In a reentrant object, these address constants (or adcons) are collected into a separate area (in AOF terminology called a *based* area) which is addressed via the sb register. When reentrant objects are linked together, the linker merges these adcon areas into a single, contiguous adcon vector, and relocates the references into the adcon vector appropriately (usually by adjusting the offset of an `LDR …,[sb,`*`offset`*`]` instruction). The output of this process is termed a *link unit*.

In this organisation, it is possible for different threadings of the code to address different static data, and for the binding of the code to its data to be delayed until execution time, (an excellent idea if the code has to be committed to ROM, even if re-entrancy is not required).

When control passes into a new link unit, a new value of sb has to be established; when control returns, the old value must be restored. A call between two separately linked program fragments is called an *inter link unit* call, or *inter-LU* call. The inter-LU calls are precisely the calls between a shared library's stub and the library's matching entry points.

Because an LDR instruction has a limited (4KB) offset, the linker packs adcons into the low-address part of the based-sb area. Currently there can be no more than 1K adcons in a client application (but this number seems adequate to support quite large programs using several Megabytes of shareable code).

The linker places the data for the inter-LU entry veneers immediately after the adcon vector (still in the based-sb area). If the stub is reentrant (to support linking into other shared libraries), then the inter-LU entry data consists of:

- the data part of the inter-LU veneer for each direct inter-LU call (which is addressed sb-relative from the separate inter-LU code part);
- the entry veneer for each address-taken library function (ie. for each function that could be invoked via a function pointer).

If the stub is not reentrant, then the inter-LU entry data is an array of function variable veneers, one for each directly exported or address-taken function in the library.

A reentrant function called via a function pointer or from a non-reentrant caller, must have its sb value loaded pc-relative, as there is no sb value relative to which to load it. This forces the entry veneer to be part of the client's private data (or there could be no re-entrancy).

6.11.3 Including data areas in a shared library

Usually, a shared library only includes areas which have both the CODE and READONLY attributes.

When you ask for a read-only copy of a data area to be included in a shared library, the linker checks it is a simple, initialised data area. The following *cannot* be included in a shared library:

- zero-initialised data (these always remain in the stub);
- COMMON data;
- stub data from the stubs of other shared libraries with which this library is being linked;
- inter-link-unit entry data and address constants.

When an area is found to be suitable for inclusion in a shared library, the following is done:

A clone of the area is created with the name `SHL$$data` and the attribute READONLY. It inherits its initialising data from the original area but it inherits no symbol definitions.

The original area is renamed `$$0` and given the attribute 0INIT. It inherits all of the symbols defined in the original area.

Area renaming is essential to ensure that multiple input areas will be sorted identically by the linker in both the stub and the shareable library, and that both the placeholder and its initialising data will be sorted into contiguous chunks. This guarantee of identical ordering—together with the absence of relocation directives—allows the placeholder to be initialised by directly copying its initialising image from the sharable library.

Names containing $$ are reserved to the implementors of the ARM software development tools, so these linker-invented area names cannot clash with any area name you choose yourself.

6.11.4 Entry veneers

The inter-LU code for a direct, reentrant inter-LU call is:

```
FunctionName
   ADD    ip, sb, #V1; calculate offset of veneer data from sb
    ADD    ip, ip, #V2
    LDMIA  ip, {ip, pc}; load new-sb and pc values
```

This allows up to 32K entry veneers to be addressed, (V1 and V2 are jointly relocated by the linker and support a word offset in the range 0-65K). The corresponding inter-LU data is:

```
    DCD    new-sb; sb value for called link unit
    DCD    entry-point ; address of the library entry point
```

Both these values are created when the stub is patched, as introduced above and described in detail on the following pages.

The inter-LU code for an indirect or non-reentrant inter-LU call is:

```
FunctionName
    ADD    ip, pc, #0      ; ip = pc+8
    LDMIA  ip, {ip, pc}    ; load new-sb and pc values
    DCD    new-sb          ; sb value for called link unit
    DCD    entry-point     ; address of the library entry
                           ; point
```

Again, the data values are created when the stub is patched.

Entry veneer initial values

The linker initialises the data part of each entry veneer as follows:

- new-sb: the index of the entry point in the array of entry points (note that the entries may not be of uniform length in the reentrant case);
- entry-point: the address of a 4-word code block, placed at the end of the inter-LU data by the linker.

Overall, an adcon/inter-LU-data area for a link unit has the layout:

```
Base                       ; sb points here
    adcons                 ; up to 1K adcons...
    inter-LU data          ; inter-LU fn-call veneer data
End
    STMFD  sp!, {r0-r6,r14}; save work registers and lr
    LDR    r0, End-4       ; load address of End
    B      |__rt_dynlink|  ; do the dynamic linking...
    DCD    Params - Base   ; offset to sb-value
Params
    parameter block        ; user-specified parameters
```

Note the assumption that a stack has been created *before* any attempt is made to access the shared library. Note also that the word preceding `End` is initialised to the address of `End`.

Entry veneer patching

A simple version of the dynamic linking code, `__rt_dynlink`, referred to above, can be implemented as described in this section.

On entry to `__rt_dynlink`, a copy of the pointer is saved to the code/parameter block at the end of the inter-LU data area, and a bound is calculated on the stub size (the entries are in index order).

```
|__rt_dynlink|
    MOV    r6, r0
    LDR    r5, [r6, #-8]   ; max-entry-index
    ADD    r5, r5, #1      ; # max entries in stub
    MOV    r4, ip          ; resume index
```

Then it is necessary to locate the matching library, which the following fragment does in a simple system-specific fashion. Note that in a library which contains no read-only static data image, r0+16 identifies the user parameter block (at the end of the inter-LU data area); if the library contains an initialising image for its static data then r0+24 identifies the user parameter block.

Here, the library location function is shown as a SWI which takes as its argument in r0 a pointer to the user parameter block and returns the address of the matching External Function Table in r0:

```
    ADD    r0, r6, #24     ; stub parameter block address
    SWI    Get_EFT_Address ; are you there?
    BVS    Botched         ; system-dependent
```

R0 now points to the EFT, which begins with the number of entries in it. A simple sanity check is that if there are fewer entries in the library than in the stub, it has probably been patched incorrectly.

```
    LDR    ip, [r0]        ; #entries in lib
    CMPS   ip, r5          ; >= #max entries in stub?
    BLT    Botched         ; no, botched it...
```

If the shared library contains data to be copied into the stub then check the length to copy:

```
    LDR    ip, [r6, #16]   ; stub data length
    BIC    ip, ip, #3      ; word aligned, I insist...
    ADD    r3, r6, #4
    LDR    r3, [r3, r5, LSL #2]; library data length
    CMPS   r3, ip
    BNE    Botched         ; library and stub lengths differ
```

Checking the stub data length and library data length match is a naive, but low-cost, way to check the library and the stub are compatible. Now copy the static data from the library to the stub:

```
    LDR    r3, [r6, #20]   ; stub data destination
    SUB    r2, r0, ip      ; library data precedes the EFT
01  SUBS   ip, ip, #4      ; word by word copy loop
    LDRGE  r1, [r2], #4
    STRGE  r1, [r3], #4
    BGE    %B01
```

Then initialise the entry vectors. First, the sb value is computed for the callee:

```
    LDR    ip, [r6, #12]   ; length of inter-LU data area
    ADD    r3, r6, #24     ; end of data area...
    SUB    r3, r3, ip      ; start of data area = sb value
```

If there is no static data in the library, #24 above becomes #16.

Then the following loop works backwards through the EFT indices, and backwards through the inter-LU data area, picking out the indices of the EFT entries which need to be patched with an `sb`, `entry-point` pair. Register Ip still holds the index of the entry that caused arrival at this point, which is the index of the entry to be retried after patching the stub. The corresponding retry address is remembered in r14, which was saved by the code fragment at the end of the inter-LU data area before it branched to `__rt_dynlink`. A small complication is that the step back through a non-reentrant stub may be either 8 bytes or 16 bytes. However, there can be no confusion between an index (a small integer) and an ADD instruction, which has some top bits set.

```
    LDR    r2, [r6, #-8]!        ; index of stub entry
00  SUB    ip, r5, #1            ; index of the lib entry
    CMPS   ip, r2                ; is this lib entry in the stub?
    SUBGT  r5, r5, #1            ; no, skip it
    BGT    %B00
    CMPS   r2, r4                ; found the retry index?
    MOVEQ  lr, r6                ; yes: remember retry address
    LDR    ip, [r0, r5, lsl #2]  ; entry point offset
    ADD    ip, ip, r0            ; entry point address
    STMIA  r6, {r3, ip}          ; save {sb, pc}
    LDR    r2, [r6, #-8]!        ; load index and decrement r6...
    TST    r2, #&ff000000        ; ... or if loaded an instr?
    LDRNE  r2, [r6, #-8]!        ; ...load index and decrement r6
    SUBS   r5, r5, #1            ; #EFT entries left...
    BGT    %B00
```

Finally, when the vector has been patched, the failed call can be retried:

```
    MOV    ip, lr                ; retry address
    LDMFD  sp!, {r0-r6, lr}      ; restore saved regs
    LDMIA  ip, {ip, pc}          ; and retry the call
```

6.11.5 Versions, compatibility and foreverness

The mechanisms described so far are very general and, of themselves, give no guarantee that a stub and a library will be compatible, unless the stub and the library were the complementary components produced by a single link operation.

Often, in systems using shared libraries, stubs are bound into applications which must continue to run when a new release of the library is installed. This requirement is especially compelling when applications are constructed by third party vendors or end users.

The general requirements for compatibility are as follows:

- a library must be at least as new as the calling stub;
- libraries can only be extended, and then only by *disjoint extension* (adding new entries to a library, or by giving to existing entries new interpretations to previously unused parameter value combinations).

In general, the compatibility of a stub and a library can be reduced to the compatibility of their respective *versions*. The ARM shared library mechanism does not mandate how versions are described, but provides an open-ended parameter block mechanism which can be used to encode version information to suit the intended usage.

Because the addresses of library entry points are not bound into a stub until run-time, the only foreverness guarantees that a library must give are:

- its entry points are in the same order in its EFT (this is a property of the shared library description given to the linker, not of the library's implementation);
- the behaviour of each exported function must be maintained compatibly between releases (beware that it is genuinely difficult to prevent users relying on unintended behaviour—the curse of *bug compatibility*).

Because a stub contains the indices of the entry points it needs, it is harmless to add new entry points to a library: the dynamic linking code simply ignores them. Of course, they must be added to the end of the list of exported functions if the first property, above, is to be maintained.

For libraries which export only code, and which make no use of static data, compatibility is straightforward to manage. Use of static data is more hazardous, and the direct export of it is positively lethal.

If a static data symbol is exported from a shared library, what is actually exported is a symbol in the library's stub. This symbol is bound when the stub is linked into an application and, from that instant onwards, cannot be unbound. Thus the direct export of a data symbol fixes the offset and length of the corresponding datum in the shared library's data area, *forever* (ie. until the next incompatible release…).

The linker does not fault the direct export of data symbols because the ARM shared library mechanism may not be being used to build a shared library, but is instead being used to structure applications for ROM. In this case a prohibition could be irksome. Those specifying or building genuine shared libraries need to be aware of this issue, and should generally not make use of directly exported data symbols.

If data must be exported directly then:

- only export data which has very stable specifications (semantics, size, alignment, etc.)
- place this data first in the library's data area, to allow all other non-exported data to change size and shape in future releases (subject to its total size remaining constant)

If a shared library makes any use of static data, it is prudent to include some unused space, so that non-exported data may change size and shape (within limits) in future releases without increasing the total size of the data area. Remember that if a *forever binary* guarantee is given, the size of the data area may never be increased.

In practice, it is rare for the direct export of static data to be genuinely necessary. Often a function can be written to take a pointer to its static data as an argument, or used to return the address of the relevant static data (thus delaying the binding of the offset and size of the datum until run-time, and avoiding the foreverness trap). It is only if references to a datum are frequent and ubiquitous that direct export is unavoidable. For example, a shared library implementation of an ANSI C library might export directly `errno`, `stdin`, `stdout` and `stderr`, (and even `errno` could be replaced by `(*__errno())`, with few implications).

6.11.6 Describing a shared library to the linker

A shared library description consists of a sequence of lines. On all lines, leading white space (blank, tab, VT, CR) is ignored.

If the first significant character of a line is:

`;` the line is ignored. Lines beginning with ';' can be used to embed comments in a shared library description. A comment can also follow a `\` which continues a parameter block description.

`>` the line gives the name and parameter block for the library. Such lines can continue over several contiguous physical lines by ending all but the last line with `\` :

```
> testlib           \ ; the name of the library image file
  "testlib"         \ ; the text name of the library ->
                    \ ; parameter block
  101               \ ; the library version number
  0x80000001
```

The first word following the > is the name of the file to hold the shared library binary image; the argument to the linker's `-Output` option is used to name the stub. Following tokens are parameter block entries, each of which is either a quoted string literal or a 32-bit integer. In the parameter block, each entry begins on a 4-byte boundary.

Within a quoted string literal, the characters `"` and `\` must be preceded by `\` (the same convention as in C). Characters of a string are packed into the parameter block in ascending address order, followed by a terminating NUL and NUL padding to the next 4-byte boundary.

An integer is written in any way acceptable to the ANSI C function `strtoul()` with a base of 0, that is, as an optional `-` followed by one of:

- a sequence of decimal digits, not beginning with 0
- a 0 followed by a sequence of octal digits
- 0x or 0X followed by a sequence of hexadecimal digits

Values which overflow or are otherwise invalid are not diagnosed.

A line beginning with a + describes input data areas to be included, read-only, in the shared library and copied at run time to placeholders in the library's clients. The general format is a list of *`object`*(*`area`*) pairs instructing the linker to include area *`area`* from object *`object`*:

```
+ object ( area ) object ( area ) ...
```

If `object` is omitted any object in the input list will match. For example:

```
+ (C$$data)
```

instructs the linker to include all areas called `C$$data`, whatever objects they are from.

If *`area`* is omitted too, then *all* suitable input data areas will be included in the library. This is the most common usage. For example:

```
+ ()
```

Finally, a ‘+’ on its own excludes all input data areas from the shared library but instructs the linker to write zero length and address or offset words immediately preceding the stub and library parameter blocks, for uniformity of dynamic linking.

All remaining non-comment lines are taken to be the names of library entry points which are to be exported, directly or via function pointers. Each such line has one of the following three forms:

`entry-name`	Names a directly exported global symbol: a direct entry point to the library, or the name of an exported datum (deprecated).
`entry-name()`	Names a global code symbol which is exported indirectly via a function pointer. Such a symbol may also be exported directly.
`entry-name(object-name)`	Names a non-exported function which is exported from the library by being passed as a function argument, or by having its address taken by a function pointer.

To clarify this, suppose the library contains:

```
void f1(...) {...}
void f2(...) {...}
static void f3(...) {...}        /* from object module o3.o */
static void (*fp2)(...) = f2;
void (*pf3)(...) = f3;
```

...and that `f1` is to be exported directly. Then a suitable description is:

```
f1
f2()
f3(o3)
pf3    /* deprecated direct export of a datum */
```

Note: `f2` and `f3` have to be listed even though they are not directly exported, so that function variable veneers can be created for them.

`f3` must be qualified by its object module name, as there could be several non-exported functions with the same name (each in a differently named object module). Note that the *module* name, not the name of the file containing the object module, is used to qualify the function name.

If `f2` were to be exported directly then the following list of entry points would be appropriate:

```
f1
f2
f2()
f3(o3)
pf3
```

Unless all address-taken functions are included in the export list, the linker will complain and refuse to make a shared library.

6.11.7 Linker pre-defined symbols

While a shared library is being constructed the linker defines several useful symbols:

`EFT$$Offset`	offset of the External Function Table from the beginning of the shared library
`EFT$$Params`	offset of the shared library's parameter block from its beginning
`$$0$$Base`	the (relocatable) address of the zero-initialised place holder in the stub
`SHL$$data$$Base`	offset of the start of the read-only copy of the data L 'from the beginning of the shared library
`SHL$$data$$Size`	size of the shared library's data section, which is also the size of the place holder in the stub

`EFT$$Offset` and `EFT$$Params` are exported to the stub and may be referred to in following link steps; the others exist only while the shared library is being constructed.

6.12 Overlays

The linker supports both static and dynamic overlays by generating tables through which calls to overlay segments are indirected. If the relevant overlay segments is not loaded, a section of code called the *overlay manager* is called to load the segment. Although the linker generates references to the overlay manager, the linker does not provide it. An object file containing the overlay manager must be supplied in the link command. For a detailed description of the overlay manager, see ➲*6.13 The Overlay Manager* on page 6-32.

Thumb: It is not possible to enter/exit an overlay segment in Thumb state. Hence all the external callable interfaces in a overlay segment should be ARM state code.

6.12.1 Static overlays

In the static case, a simple 2-dimensional overlay scheme is supported. There is one root segment, and as many memory partitions as specified by the user (for example, 1_, 2_, etc.). Within each partition, some number of overlay segments (for example, 1_1, 1_2, …) share the same area of memory. You specify the contents of each overlay segment and the linker calculates the size of each partition, allowing sufficient space for the largest segment in it.
All addresses are calculated at link time so statically overlaid programs are not relocatable.
A hypothetical example of the memory map for a statically overlaid program might be:

2_1	2_2	2_3		high address
1_1	1_2	1_3	1_4	
root segment				low address

Segments 1_1, 1_2, 1_3 and 1_4 share the same area of memory. Only one of these segments can be loaded at any given instant; the remainder must be on backing store.

Similarly, segments 2_1, 2_2 and 2_3 share the 2_ area of memory, but this is entirely separate from the 1_ partition.

It is a current restriction that an overlay segment name is of the form `partition_segment` and contains 10 or fewer characters. Note that there is no requirement for `partition` and `segment` to be numeric as shown in the example: any alphanumeric characters are acceptable.

6.12.2 Dynamic overlays

A dynamic or relocatable overlay scheme is obtained by specifying the `-Relocatable` command-line option. In this case:

- the root segment is a (load-time) relocatable AIF image;
- each overlay segment is a plain binary image with relocation directives appended.

When using relocatable overlays, it is expected that:

- the overlay manager will allocate memory for a segment when it is first referenced;
- a segment will be unloaded, and the memory it occupies freed, by an explicit call to the overlay manager.

Here, you give each overlay segment a simple name (no embedded underscore), and let the linker link each as if it were in its own partition (dynamically allocated by the overlay manager).

Nevertheless, if a two-dimensional naming scheme is used, the linker will generate segment clash tables (see below), and segments can be unloaded implicitly by the overlay manager when a clashing segment is loaded. In effect, this supports the classification of dynamic overlay segments into disjoint sets of *not co-resident* objects.

A dynamic overlay segment (including a root segment) is followed by a sequence of relocation directives. The sequence is terminated by a word containing –1. Each directive is a 28-bit byte offset of a word or instruction to be relocated, together with a flag nibble in the most significant 4 bits of the word. Flag nibbles have the following meanings:

0	relocate a word in the root segment by the difference between the address at which the root was loaded and the address at which it was linked
1	relocate a word in an overlay segment by the address of the root
2	relocate a word in an overlay segment by the address of the segment
3	relocate a B or BL from an overlay segment to the root segment, by the difference (in words) between the segment's address and the root's address
7	relocate a B or BL from the root segment to an overlay segment, by the difference (in words) between the root's address and the segment's address, (such relocation directives always refer to a PCIT entry in an overlay segment, which is used to initialise a PCIT section in the root when the overlay segment is loaded; see ➲*6.13 The Overlay Manager* on page 6-32 for further explanation)

6.12.3 Assignment of AREAs to overlay segments

The linker assigns AOF AREAs to overlay segments under user control (see below). Usually, a compiler produces one code AREA and one data AREA for each source file (called C$$code and C$$data when generated by the C compiler). The C compiler option `-ZO` allows each separate function to be compiled into a separate code AREA, allowing finer control of the assignment of functions to overlay segments, (but at the cost of slightly enlarged code and enlarged object files). The user controls the overlay structure by describing the assignment of certain AREAs to overlay segments. For each remaining AREA in the link list, the linker will act as follows:

- if all references to the AREA are from the same overlay segment, the AREA is included in that segment, otherwise
- the AREA is included in the root segment.

This strategy can never make an overlaid program use more memory than if the linker put all remaining AREAs in the root, but it can sometimes make it smaller.

By default, only code AREAs are included in overlay segments. Data AREAs can be forcibly included, but it is the user's responsibility to ensure that doing so is meaningful and safe.

On disc, an overlaid program is organised as a directory containing a root image and a collection of overlay segments. The name of the directory is specified to the Linker as the argument to its `-Output` flag.

The linker creates the following components within the application directory:

root segment	`root,` which is an AIF image
overlay segments	plain binary image fragments, for example:

```
1_1
1_2
...
2_1
...
```

6.12.4 Describing an overlay structure to the linker

The overlay file, named as argument to the `-OVerlay` option, describes the required overlay structure. It is a sequence of *logical lines*:

- '\' immediately before the end of a physical line continues the logical line on the next physical line;
- any text from ';' to end of the logical line (inclusive) is a comment

Each logical line has the following structure:

```
segment-name ["(" base-address ")" ]
      module-name [ "(" list-of-AREA-names ")" ]
```

where

base-address	is the address of the segment. The value can be expressed in decimal or hexadecimal. For example: 12 (decimal) 0x1FF0 (hexadecimal) &20C0 (hexadecimal)
list-of-AREA-names	is a comma-separated list. If omitted, all AREAs with the CODE attribute are included.
module-name	is either the name of an object file (with all leading pathname components removed), or the name of a library member (with all leading pathname components removed).

In the following example, `sort` would match the C library module of the same name:

```
1_1    edit1 edit2 editdata(C$$code,C$$data) sort
```

Note: These rules require that modules have unique names within a link list. For example, it is not possible to overlay a program made up from `test/thing.o` and `thing.o` (two modules called thing). This is a restriction on overlaid programs only.

To help partition a program between overlay segments, the linker can generate a list of inter-AREA references. This is requested by using the `-Xref` option. In general, if area A refers to area B, for example because `fx` in area A calls `fy` in area B, then A and B should not share the same area of memory. Otherwise, every time `fx` calls `fy`, or `fy` returns to `fx`, there will be an overlay segment swap.

The `-MAP` option requests the linker to print the base address and size of every AREA in the output program. Although not restricted to use with overlaid programs, `-MAP` is most useful with them, as it shows how AREAs might be packed more efficiently into overlay segments.

Even though segments can be placed at specific memory locations by supplying base addresses, clash detection relies only on the names. For example, if 1_1 is placed at 0x8000, and 1_2 is placed at 0x10000, and 1_1 does not overlap 1_2, then the linker will still 'believe' they will clash, because they have the same partition name.

6.13 The Overlay Manager

This section describes in detail how a static overlay manager operates. The details of a dynamic overlay manager are very similar. In both cases, details specific to the target system are omitted.

The job of the overlay manager is to load, swap, and unload overlay segments. This is done by trapping inter-segment function calls. References to data are resolved statically by the linker when each overlay segment is created. De-referencing a datum cannot cause an overlay segment to be loaded.

Every inter-segment procedure call is indirected through a table in the root segment that traps unloaded target overlay segments, (the *procedure call indirection table*, or PCIT). PCITs are created by the linker. Each overlay segment contains the data required to initialise its section of the PCIT when it is loaded. This is simply a table of Branch instructions, one for each function exported by the overlay segment. As the linker knows the locations of each segment of the PCIT and of each function exported by each overlay segment, it can create these Branch instructions at link time. (In a dynamic overlay scheme, all segments, including the root, are assumed to be linked at 0, and a type 7 relocation directive is generated to describe the relocation of each of the initializing branch instructions).

Initially, every sub-section of the procedure-call indirection table (PCIT) in the root segment is initialised as follows (one for each procedure exported by the corresponding overlay segment.):

```
STR LR,[PC, #–8]
```

A call to an entry in the root PCIT overwrites that entry, and every following entry, with the return address, until control falls off the end of that section of the PCIT into code which:

- finds which entry was called
- loads the corresponding overlay segment (and executes its relocation directives, if it is relocatable)
- overwrites the PCIT subsection with the associated branch vector (from the just-loaded overlay segment)
- retries the call

Future calls to this section of the PCIT will encounter instructions of the form `B` *`fn`*, adding only a few cycles to the procedure call overhead. This will persist until some function call, function return, or explicit call to the overlay manager causes this PCIT segment to be overwritten.

The load-segment code not only loads an overlay, but also re-initialises the PCIT sections corresponding to segments with which it cannot co-reside. It also installs handlers to catch returns to segments which have been unloaded.

The linker generates references to two symbols which must be defined by the overlay manager:

`Image$$Overlay_init`	initialises the overlay manager. This is done automatically when executing an AIF file.
`Image$$load_seg`	handles the loading of segments. This is called from unloaded segments' PCIT sections.

6.13.1 The structure of a PCIT section

The per-PCIT-section code and data structures are shown immediately below. These are created by the linker and used by the overlay manager. They are justified and explained in the following subsections. The space cost of this code is (9 + #Clashes + #Entries) words per overlay segment. Most of the work is done in the function `Image$$load_seg` (which is shared between all PCIT sections), and in `load_segment` (which is the common tail for both `Image$$load_seg` and `load_seg_and_ret`). For an explanation of `load_seg_and_ret` see ❯*6.13.4 Intercepting returns to overwritten segments* on page 6-37.

```
        STR    LR, [PC, #-8]            ; guard word
EntryV  STR    LR, [PC, #-8]            ; > one entry for each
        ...                             ; > procedure exported
        STR    LR, [PC, #-8]            ; > by this overlay segment
        BL     Image$$load_seg
PCITSection
Vecsize DCD    .-4-EntryV               ; size of entry vector
Base    DCD    ...                      ; initialised by the linker
Limit   DCD    ...                      ; initialised by the linker
Name    DCB    11 bytes                 ; 10-char segment name + NUL
Flags   DCB    0                        ; ...and a flag byte
ClashSz DCD    PCITEnd-.-4              ; size of table following
Clashes DCD    ...                      ; >table of pointers or
offsets
              ...                       ; >to segments which cannot
        DCD    ...                      ; >co-reside with this one
PCITEnd
```

Pointers to clashing segments point to the appropriate `PCIT`*`Section`* labels (ie. into the middle of PCIT sections).

(If the overlays are relocatable, offsets between `PCIT`*`Section`* labels are used rather than addresses which would themselves require relocation.)

We now define symbolic offsets from PCITSections for the data introduced here. These are used in the `Image$$load_seg` code described in the next subsection.

```
O_Vecsize   EQU Vecsize-PCITSection
O_Base      EQU Base-PCITSection
O_Limit     EQU Limit-PCITSection
O_Name      EQU Name-PCITSection
O_Flags     EQU Flags-PCITSection
O_ClashSz   EQU ClashSz-PCITSection
O_Clashes   EQU Clashes-PCITSection
```

6.13.2 The Image$$load_seg code

The `Image$$load_seg` code contains a register save area which is shared with `load_seg_and_ret`. Both of these code fragments are veneers on `load_segment`. Both occur once in the overlay manager, not once per segment. Note that the register save area could be separated from the code and addressed via an address constant, as `ip` is available for use as a base register. Note also that `load_segment` and its veneers preserve `fp`, `sp`, and `sl`, which is vital.

```
    STRLR  STR LR, [PC, #-8]            ; a useful constant
    Rsave  %   10*4                     ; for R0-R9
    LRSave %   4
    PCSave %   4
Image$$load_seg
    STR    R9, RSave+9*4                ; save a base register...
    ADR    R9, RSave
    STMIA  R9, {R0-R8}                  ; and some working registers
    MRS    R8,CPSR
    STR    R8,PSRSave
    LDR    R0,[LR,#-8]
    STR    R0, LRSave                   ; ...save it here ready for
                                        ; retry
    LDR    R0, STRLR                    ; look for this...
    SUB    R1, R8, #8                   ; ...starting at penultimate
                                        ; overwrite
01  LDR    R2, [R1, #-4]!
    CMP    R2, R0                       ; must stop on guard word...
    BNE    %B01
    ADD    R1, R1, #4                   ; gone one too far...
    STR    R1, PCSave                   ; where to resume
    B      load_segment                 ; ...and off to the common tail
```

On entry to `load_segment`, R9 points to a register save for {R0-R9, LR, PC}, and R8 identifies the segment to be loaded. FP, SP and SL are preserved at all times by the overlay segment manager. There is only one copy of `Image$$load_seg`, shared between all PCIT sections. A similar section of code, called `load_seg_and_ret`, is invoked on return to an unloaded segment (see *6.13.4 Intercepting returns to overwritten segments* on page 6-37). This code is also a veneer on `load_segment` which shares RSave, LRSave and PCSave, and which branches to load_segment with R8 and R9 set up as described above.

Note: The code for STR LR, [PC, #-8] is 0xE50FE008. This address is unlikely to be in application code space, so overwriting indirection table entries with an application's return addresses is safe.

6.13.3 The load_segment code

`Load_segment` must:

- re-initialise the global PCIT sections for any overlay segment which ‘clashes’ with this one, while checking the stack for return addresses that are invalidated by so doing, and installing return handlers for them
- allocate memory for the about-to-be-loaded segment, (if the overlay scheme is dynamic)—this is system-specific
- load the required overlay segment (system-specific)
- execute the loaded segment’s relocation directives (if any)
- copy the overlay segment’s PCIT into the global PCIT
- restore the saved register state (with pc and lr suitably modified)

On entry to `load_segment`, R9 points to the register save area, and R8 to the PCIT section of the segment to load. First the code must re-initialise the PCIT section (if any) which clashes with this one:

```
load_segment
    ADD    R1, R8, #O_Clashes
    LDR    R0, [R8, #O_ClashSz]
01  SUBS   R0, R0, #4
    BLT    Done_Reinit               ; nothing left to do
    LDR    R7, [R1], #4              ; a clashing segment...
    ADD    R7, R7, R8                ; only if root is relocatable
    LDRB   R2, [R7, #O_Flags]
    CMPS   R2, #0                    ; is it loaded?
    BEQ    %B01                      ; no, so look again
    MOV    R0, #0
    STRB   R0, [R7, #O_Flags]        ; mark as unloaded
    LDR    R0, [R7, #O_Vecsize]
    SUB    R1, R7, #4                ; end of vector
    LDR    R2, STRLR                 ; init value to store...
 02 STR    R2, [R1, #-4]!            ;>
    SUBS   R0, R0, #4                ;> loop to initialise the
                                     ; PCIT segment
    BGT    %B02                      ;>
```

Next, the stack of call frames for return addresses invalidated by loading this segment is checked, and handlers are installed for each invalidated return. This is discussed in detail in the next subsection. Note that R8 identifies the segment being loaded, and R7 the segment being unloaded.

```
BL     check_for_invalidated_returns
```

Segment clashes have now been dealt with, as have the re-setting of the segment-loaded flags and the intercepting of invalidated returns. It’s now time to load the required segment. This is system specific, so the details are omitted (the name of the segment is at offset O_Name from R8).

On return, calculate and store the real base and limit of the loaded segment and mark it as loaded:

```
    BL     _host_load_segment          ; return base address in R0

    LDR    r4, [r8, #PCITSect_Limit]
    LDR    r1, [r8, #PCITSect_Base]
    SUB    r1, r4, r1                  ; length
    STR    r0, [r8, #PCITSect_Base]    ; real base
    ADD    r0, r0, r1                  ; real limit
    STR    r0, [r8, #PCITSect_Limit]
    MOV    r1, #1
    STRB   r1, [r8, #PCITSect_Flags]   ; loaded = 1
```

The segment’s entry vector is at the end of the segment; it must be copied to the PCIT section identified by R8, and zeroed in case it is in use as zero-initialised data:

```
    LDR    r1, [r8, #PCITSect_Vecsize]
    ADD    r0, r0, r1                  ; end of loaded segment...
    SUB    r3, r8, #8                  ; end of entry vector...
    MOV    r4, #0                      ; for data initialisation
01  LDR    r2, [r0, #-4]!              ;> loop to copy
    STR    r4, [r0]                    ; (zero-init possible data
                                       ;  section)
    STR    r2, [r3], #-4               ;> the segment's PCIT
    SUBS   r1, r1, #4                  ;> section into the
    BGT    %B01                        ;> global PCIT...
```

Finally, continue execution:

```
    LDR    R0,PSRSave
    MSR    CPSR,R0
    LDMIA  R9,{R0,R9,LR,PC}
```

6.13.4 Intercepting returns to overwritten segments

The overlay scheme described so far is sufficient, provided no function call unloads any overlay in the current call chain. As a specific example, consider a root segment and two procedures, A and B in overlays 1_1 and 1_2 respectively. Note that A and B may not be co-resident. Then any pattern of calls like:

```
((root calls A, A returns)* (root calls B, B returns)*)*
```

is unproblematic. However, A calls B is disastrous when B tries to return (as B will return to a random address within itself rather than to A).

To fix this deficiency, it is necessary to intercept (some) function returns. Trying to intercept all returns would be expensive; at the point of call there are no working registers available, and there is nowhere to store a return address, (the stack cannot be used without potentially destroying the current function call's arguments). The following observations describe an efficient implementation:

- a return address can only be invalidated by loading a segment which displaces a currently loaded segment
- at the point that a segment is loaded, the stack contains a complete record of return addresses which might be invalidated by the load

Before loading a segment, check the procedure call back-trace (including the value stored in LRSave) for return addresses which fall in the segment about to be overwritten. Replace each such return address by a pointer to a return handler which loads the segment before continuing the return.

There is no simple way to avoid using a fixed pool of return handlers. You cannot use the stack (in a language-independent manner) because its layout is only partly defined in mid function call. You could use a variant of the language-specific stack-extension code, but it would complicate the implementation significantly, and make some aspects of the overlay mechanism language specific. Similarly, it would be unwise to make any assumptions about the availability or management of heap space.

Using a fixed pool of handlers is not as bad as it first seems. A handler can only be needed if a call overwrites the calling segment. If this is done strictly non-recursively (meaning that if any P in segment 1 calls some Q in segment 2, then no R in segment 2 may call any S in segment 1 until Q has returned), the number of handlers required is bounded by the number of overlay segments. If recursive calls are made between overlay segments, performance will be very poor unless a large amount of work is done by each call. It is hard to envisage an application which would require an unbounded depth of recursion, and would perform significant amounts of work at each level (a recursively invokable CLI is such an example, but in this case it is hard to see why a moderate fixed limit on the depth of recursion would be unacceptable).

Note: Only the most recent return should be allocated a return handler. For example, assume that there is a sequence of mutually recursive calls between segments A and B, followed by a call to C which unloads A. Only the latest return to A needs to be trapped, because as soon as A has been re-loaded the remainder of the mutually-recursive returns can unwind without being intercepted.

6.13.5 Return handler code

A return handler must store the real return address, the identity of the segment to return to (eg. the address of its PCIT section), and it must contain a call (indirectly) to the load_segment code. In addition, it is assumed that the handler pool is managed as a singly linked list. Then the handler code is:

```
        BL     load_seg_and_ret
RealLR  DCD    0; space for the real return address
Segment DCD    0; -> PCIT section of segment to load
Link    DCD    0; -> next in stack order
```

`RealLR`, `Segment` and `Link` are set up by `check_for_invalidated_returns`.

6.13.6 The load_seg_and_ret code

`HStack` and `HFree` are set up by `overlay_mgr_init`, and maintained by `check_for_invalidated_returns`. For simplicity, they are shown here as PC-relative-addressable variables. More properly, they are part of the data area shared with `Image$$load_seg`. As already noted, this data area can be addressed via an address constant, as ip is available as a base register.

```
HStack  DCD0              ; top of stack of allocated handlers
HFree   DCD0              ; head of free-list

load_seg_and_ret
    STR    R9, RSave+9*4  ; save a base register...
    ADR    R9, RSave
    STMIA  R9, {R0-R8}    ; … and some working registers
    MSR    R8,CPSR
    STR    R8,PSRSave
    LDMIA  LR,{R0,R1,R2}
    STR    R0, LRSave
    STR    R0, PCSave
; Now unchain the handler and return it to the free pool
; (by hypothesis, HStack points to this handler...)
    STR    R2, HStack     ; new top of handler stack
    LDR    R2, HFree
    STR    R2, [R8, #8]   ; Link -> old HFree
    SUB    R2, R8, #4
    STR    R2, HFree      ; new free list
    MOV    R8, R1         ; segment to load
    B      load_segment
```

6.13.7 The check_for_invalidated_returns code

This code loads the segment identified by R8 into the slot identified by R7 to check LRSave and the chain of call-frames for the first invalidated return address. R7-R9, FP, SP and SL must be preserved.

```
    ADR    R6, LRSav       ; 1st location to check
    MOV    R0, FP          ; temporary FP...
01  LDR    R1, [R6]        ; the saved return address...
    LDR    R2, [R7, #O_Base]
    CMPS   R1, R2          ; see if >= base...
    BLT    %F02
    LDR    R2, [R7, #O_Limit]
    CMPS   R1, R2          ; ...and < limit
    BLT    FoundClash
02  CMPS   R0, #0          ; bottom of stack?
    MOVEQ  PC, LR          ; yes => return
    SUB    R6, R0, #4
    LDR    R0, [R0, #-12]  ; previous FP
    B      %B01
```

A handler is allocated for a segment containing a return address invalidated by the segment load:

```
FoundClash
    LDR    R0, HFree       ; head of chain of free handlers
    CMPS   R0, #0
    BEQ    NoHandlersLeft
                           ; Transfer the next free handler to head
                           ; of the handler stack.
    LDR    R1, [R0, #12]   ; next free handler
    STR    R1, HFree
    LDR    R1, HStack      ; the active handler stack
    STR    R1, [R0, #12]
    STR    R0, HStack      ; now with the latest handler linked in
                           ; Initialise the handler with a BL
                           ; load_seg_and_ret, RealLR and Segment.
    ADR    R1, load_seg_and_ret
    SUB    R1, R1, R0      ; byte offset for BL in handler
    SUB    R1, R1, #8      ; correct for PC off by 8
    MOV    R1, R1, ASR #2  ; word offset
    BIC    R1, #&FF000000
    ORR    R1, #&EB000000  ; code for BL
    STR    R1, [R0]
    LDR    R1, [R6]
    STR    R6, [R0, #4]    ; RealLR
    STR    R0, [R6]        ; patch stack to return to handler
```

```
        STR    R7, [R0, #8]    ; segment to re-load on return
        MOVS   PC, LR          ; and return
NoHandlersLeft
        ...                    ; initial creation of handler pool omitted
                               ; for brevity.
```

6.14 Scatter Loading

6.14.1 Introduction

Scatter loading 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 the regions to be loaded into memory at addresses different to their execution addresses. For example, initialised read/write data can be loaded into ROM but it will need to be copied to RAM when the program is executing. You can specify regions which will act as overlays.

6.14.2 Definitions

Execution regions

The memory used by a program when it is executing can be split into a set of disjoint regions each of which is contiguous chunk of memory. These regions are called Execution Regions.

Load regions

The memory used by a program before it starts executing but after it has been loaded into memory can also be split into a set of disjoint regions. These regions are called Load Regions.

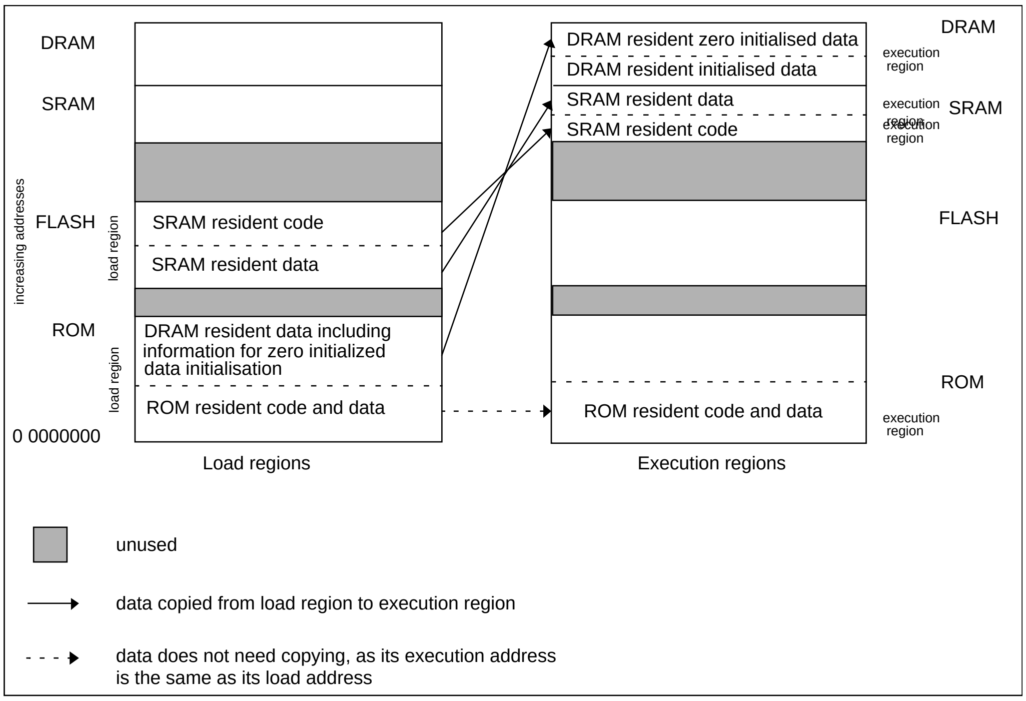

DRAM
SRAM
FLASH
ROM
0 0000000
increasing addresses
load region
SRAM resident code
SRAM resident data
load region
DRAM resident data including information for zero initialized data initialisation
ROM resident code and data
Load regions
DRAM resident zero initialised data
DRAM resident initialised data
SRAM resident data
SRAM resident code
ROM resident code and data
Execution regions
execution region
DRAM
SRAM
FLASH
ROM
unused
data copied from load region to execution region
data does not need copying, as its execution address is the same as its load address

Figure 6-1: Simple example of scatter loading

See 6.2.3 *Scatter loading command-line options* on page 6-7 for details of the command-line options.

6.14.3 Scatter loading description format

The file format reflects the hierarchy of object areas, execute regions and object areas. An object area can be in precisely one execution region. An execution region can be in precisely one load region.

Comments are any text from a semicolon (;) to the end of the line.

Load Region Specification

The file is a list of load region specifications. These are of the form

```
load_region base_address [ region_size_limit ] {
      <list of execution regions>
}
```

where:

`load_region`	is the name of the load region. This is used to name the output file generated for this load region. The maximum length of the name is the maximum length of a filesystem directory entry or 31 characters, whichever is smaller.
`base_address`	is a number. The number can be expressed in decimal or hexadecimal using either the 0x or & formats. So &1234ABCD, 0x1e3 and 1 are acceptable but 1234CD is not. Base addresses must be word-aligned.
`region_size_limit`	is optional. If this parameter specified, an error will result if the region exceeds this size.

6.14.4 Execution region specification

An execute region is described by a similar looking construct:

```
exec_name base_address [ OVERLAY ] {
      <white space separated list Of Object Area Specifications>
}
```

The `exec_name` is a convenient label. When the OVERLAY option is specified, `exec_name` is used as an overlay name. Hence it will be limited to 10 characters in length in that case. Otherwise the length is limited to 31 characters.

The base address of the execution region must be specified.

Object area specifications

The object area specifications are similar to the existing object area specifications used in the existing overlay scheme. The various forms are

`module_name`	is an object file for object library name with leading pathname components removed. The list of object area specifications can be separated by newlines
`module_name`	comma-separated list of AREA names
`module_name`	comma-separated list of AREA attribute selections
`*`	comma-separated list of AREA attribute selections. This form allows all AREAs in all modules not already specified with the specified attributes to be placed in an execution region.

The AREA attribute selections are `+RW`, `+RO` and `+ZI`. These select areas with those attributes in this module to be placed in this execution region:

`+RO`	selects read only areas
`+ZI`	selects zero-initialised data areas
`+RW`	selects any area which not zero initialised or read only.

Just specifying a module name is equivalent to:

```
module_name ( +RO )
```

Root region specification

In the overlay scheme (➲*6.12 Overlays* on page 6-28), the root segment is used to absorb all the AREAs which are not put into an overlay segment in the overlay description.The root segment also contains the entry point and initialisation code.The root segment code base is specified by the -RO-base linker option. and the read/write base is specified by the -RW-base option.

In the scatter loading scheme, a root load region performs a similar role. The root load region contains two parts - the root read only execution region and the root read/write execution region.

The root load region can contain the following:

- the entry point
- the root PCIT and a table that relates overlay load addresses to PCIT section addresses
- the information needed for regions to be copied from their load addresses to their execution addresses
- the debug information

As in the overlay scheme, any areas unspecified in the description file will be placed in the root load region. This includes the THUMB interworking veneers and the overlay manager workspace space area allocated by the linker.

There is special form of Load Region Specification for the root load region. If one were to be strict about this, the root data is an execution region which is contained within a load region. However there are some characteristics of the root region which make a full specification of the root load and execution regions unnecessarily verbose.

The root region read-only AREAs have load addresses identical to their execution addresses. No overlays can be placed in the root region. Also any AREAs not appearing in any execution region in the scatter load description file will be placed in the root execution region appropriate to the AREA's read only attribute.

The simplified form of specification for the root load and execution regions is:

```
ROOT base_address [ region_size_limit ]
```

This sets the base address for the root load region and, hence, the base address for the root read only execution region. This execution region must have its load address equal to its execution address as it contains the application entry point.

```
ROOT-DATA base_address
```

This sets the base for the root read/write execution region.

Both addresses must specified somewhere in the scatter loading description file.

When the `-bin` option is specified the root load region will be placed in a file called `root` in the output directory.

6.14.5 Implications

It is possible to specify overlapping execution regions. Overlapping execution regions are allowed but only if all the overlapping execution regions are specified as overlays in the scatter load description file. Otherwise overlapping load regions are detected and flagged as errors. There is also one restriction on overlapping overlays. The linker uses the same mechanism to detect clashing overlay regions as it does to detect clashing overlay segments in the overlay scheme. So the overlay region names are expected to be of the form

```
partition_segment
```

Overlaid execution regions with the same partition name are deemed to clash by the linker. If the linker detects two overlaid execution regions which do not have the same partition name, the error message:

```
Overlaid regions <region1> and <region2> clash unexpectedly
```

will be generated. <*region1*> and <*region2*> are replaced by the names of the clashing execution regions.

A user can specify a size limit for a load region. If the size of a load region exceeds this value an error will be flagged.

The user can position execution regions easily. However if the user wishes to have a set of execution regions continuous in memory when executing, there is no mechanism in the scatter loading description file to specify this.

6.14.6 Example

Consider the following memory map.

```
Base          Size          Name
0x0           0x8000        ROM
0x8000        0x8000        SRAM
0x20000       0x40000       EEPROM
0x100000      0x100000      DRAM
```

The application consists of five object files:

```
init.o obj1.o obj2.o obj3.o obj4.o
```

obj4 requires a large zero initialised data area. This is to be placed in DRAM. The remaining read/write data areas are to be placed in SRAM along with the code from obj3.o. obj2.o and init.o are to be placed in the root.

```
ROOT 0x0  ; Specify the root region base address.

ROOT-DATA 0x8000
EEPROM 0x20000 0x40000 {
       EEPROM 0x20000 {
              obj1.o(+RO)   ; Read Only AREAs to execute in the EEPROM.
              obj4.o(+RO)   ; This region has the same execution address
                            ; as its load address.
       }
       SRAM 0x9000  {
              obj3.o
              obj1.o(+RW,+ZI) obj4.o(+RW)
       }
       DATA 0x100000 { obj4.o(+ZI) }
}
```

Notes

1. AREAs that need to be grouped together in memory must be put in the same execution region.
2. Execution regions which have the same load and execution addresses need to be placed first in the execution region.

6.14.7 Overlay handling

If overlays are present, the root PCIT will be generated along with the information necessary to copy the PCIT to a read write execution region.

A table of information that will enable an overlay manager to perform overlay paging will be generated. This table will be referred to by a linker generated symbol: `Root$$OverlayTable`. The first word in the table is the number of entries in the table. The table will be a sequence of entries each 3 words long. There will be one entry per overlay segment. Each entry is:

Word 0	Segment length in bytes
Word 1	Execution Address of the PCIT section for this overlay.
Word 2	Load Address of the overlay.

Overlays cannot be put into the root load region.

The linker will generate references to the symbols `Image$$overlay_init` and `Image$$load_seg`. `Image$$load_seg` refers to the routine used to move segments to their execution addresses when 'paged in'.

6.14.8 Initialisation

Symbol generation

Symbols will be generated for each non overlaid execution region with a load address which differs from its execution address. For zero initialised data, the symbols will be of the form

```
Image$$reg$$ZI$$Base
Image$$reg$$ZI$$Length
```

where `reg` is the execution region name.

The linker will sort AREAs within execution regions in the same order as for an AIF image. Hence non zero initialised data that needs copying will be contiguous.

For other areas the symbols are

`Load$$reg$$Base`	Load address of the region
`Image$$reg$$Base`	Execution address of the region.
`Image$$reg$$Length`	Execution region length in bytes (multiple of 4 bytes)

where `reg` is the execution region name.

A user could construct an initialisation routine using these symbols. The root PCIT will be in the root read/write execution region and will be included in the copying operation for the root data.

In the example above, we could have the initialisation expressed in the following pseudo C.

```
void Init(void)
{
      __copy(Image$$root$$Base,Load$$root$$Base,Image$$root$$Length);
      __copy(Image$$SRAM$$Base,Load$$SRAM$$Base,Image$$SRAM$$Length);
      __zero(Image$$root$$ZI$$Base, Image$$root$$ZI$$Length);
      __zero(Image$$SRAM$$ZI$$Base, Image$$SRAM$$ZI$$Length);
      __zero(Image$$DRAM$$ZI$$Base, Image$$DRAM$$ZI$$Length);
}
```

where __copy performs a fast copy in the manner of memcpy, and __zero zeroes the specified number of bytes.

The initialisation function will need to be called before the application main program is entered. For example:

```
       ENTRY __main
       EXPORT __main
       IMPORT Init
       IMPORT __entry
__main
       BL Init
       BL __entry
       END
```

7 Symbolic Debugger

This chapter describes the ARM Symbolic Debugger and its command language.

Symbolic Debugger

7.1 About armsd

`armsd` 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. armsd is normally used to run ARM Image Format images.

See ➲*Chapter 2, C Compiler*, ➲*Chapter 3, Assembler* and ➲*Chapter 6, Linker* for more information on generating debugging data.

7.2 Line-Speed Negotiation

The debugger will attempt to operate at the configured line speed (9600, 19200 or 38400 baud). When first invoked, and whenever it detects that the debuggee has been reset, the debugger operates at 9600 baud. Similarly, when the debug monitor (demon) is reset, it operates at 9600 baud. One of its first acts after reset is to send a request to the debug monitor (at 9600 baud) to start operating at the configured (higher) line rate.

The default configured line speed is:

SunOS	38400 baud
PC/DOS	19200 baud
Macintosh	19200 baud

This can be re-configured after installation using the reconfig utility (see ➲*Chapter 12, ARM Tool Reconfiguration Utility*).

7.3 Command-line Options

You invoke `armsd` using the command:

```
armsd {options} image-name {arguments}
```

The options are listed below. Upper-case is used to show the allowable abbreviations. The options must go before the image name. Anything after the image name is treated as a command-line option.

`-Help`	Gives a summary of the armsd command-line options
`-Size n`	Specifies the memory size required for the image being debugged. *n* may be prefixed by 0x for hex values, and suffixed with either K or M to specify KB or MB respectively. The result of this option depends on the user-supplied memory model linked in with the debugger. (See also ➲*14.8 Memory Models* on page 14-6.)
`-Little`	Specifies that memory should be little endian
`-Big`	Specifies that memory should be big endian
`-SErial`	Specifies that the debugger should act as a front end to the Platform Independent Evaluation Card
`-Armul`	Specifies that the debugger should act as a front end to the software ARM emulator, ARMulator
`-Port n`	Specifies whether the first or second serial port should be used
`-LINEspeed n`	Specifies the linespeed for communication through the serial port: only the values 9600, 19200 and 38400 are permitted
`-Processor`	Specifies the cpu type
`-nofpe` or `-fpe`	Specifies whether the ARMulator should load the FPE on start-up. (By default, the FPE is loaded, to match the Demon environment.) When testing code compiled using the floating-point library, you may wish not to load the FPE. See ➲*Chapter 16, Software Floating Point* for more information on the floating-point library.
`–oldexpressions`	Specifies that commands which accept low-level symbols by default should revert to their pre-version 2.0 behaviour
`–newexpressions`	Specifies that commands which accept low-level symbols by default should follow the version 2.0 behaviour
`-Clock`	Specifies the clock speed for the ARMulator. (➲*7.15 Performance simulation using armsd* on page 7-35)

Note: Many of these options can be configured as the default, so you do not need to specify them. See ➲*Chapter 12, ARM Tool Reconfiguration Utility.*

7.3.1 Extensions to armsd for EmbeddedICE

The following extensions have been added for use with EmbeddedICE

- Command-line options
- `armsd` commands
- `armsd` internal variables

Command-line options

`-SErial`	Specifies that `armsd` should use a serial based RDP driver. This is used to communicate with the EmbeddedICE board via the serial port only.
`-SERPAR`	Specifies that `armsd` should use both the serial and parallel ports for RDP communication. This is used to communicate with the EmbeddedICE board via both serial and parallel ports.
`-Reset`	Holds the ARM's reset pin HIGH while it is forcing the debuggee to halt execution. This can be used to ensure that the ARM does not execute any code before `armsd` takes control.
`-SPort` *`n`*	Specifies which serial port should be used.You can set this option to 1 or 2. When using a PC, you can specify an address to be used as the port address.
`-PPort` *`n`*	Specifies which parallel port should be used.You can set this option to 1 or 2. When using a PC, you can choose to specify an address to be used as the port address.
-`LOadconfig` *`name`*	Specifies the file containing the configuration data to be loaded
-`Selectconfig` *`name`*	Specifies the target configuration to use

`armsd` commands

`listconfig`		Lists the configurations known to the debug agent	
`loadagent` *`file`*		Downloads a replacement EmbeddedICE ROM image, and starts it (in RAM)	
`loadconfig` *`file`*		Loads an EmbeddedICE configuration data file.	
`selectconfig` *`name version`*		Selects an EmbeddedICE configuration to use:	
	`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`	Must use version n
		`n+`	Must use version n or later

The highest-numbered version meeting the *`version`* constraint is used. For more information see ➲*18.3 Configuration Data* on page 18-2.

`armsd` internal variables

`$icebreaker_lockedpoints`	Shows or sets locked EmbeddedICE macrocell points
`$semihosting_enabled`	Enables semihosting
`$semihosting_vector`	Sets up semihosting SWI vector

For more information about using these variables, see ➲*7.16 Semihosting under EmbeddedICE* on page 7-41 and ➲*18.4 Accessing the EmbeddedICE Macrocell Directly* on page 18-3.

7.4 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 2, C Compiler*, ➲*Chapter 3, Assembler* and ➲*Chapter 6, Linker*. Examples that demonstrate running programs under the command-line-based `armsd` are given in the manual, *Software Development Toolkit Programming Techniques*.

plain text	Plain text in this form should be typed in as is.
`typewriter`	Command syntax patterns are given throughout to show you what you should type to achieve an effect.
`oblique 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 in that command. 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.

7.5 Specifying Source-level Objects

7.5.1 Variable names and context

It is often sufficient to refer to variables by their names in the original source code. To print the value of a variable, simply 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 will also give 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 this is likely to lead to 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.

The way to refer to variable within a particular activation of a function is:

```
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
```

Although the syntax of variables and contexts is complex, in practice the various syntax extensions needed to differentiate between different objects rarely have to be used together.

7.5.2 Program locations

Some commands require arguments that refer to locations in the program. You can refer to a place in the program by:

- procedure entry/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 *7.11.1 Low-level symbols* on page 7-23 for further details.

Procedure entry/exit

Using a procedure name alone sets a breakpoint (see *7.9 Controlling Execution* on page 7-17) 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, notably when a file is included within a function. To resolve any ambiguities, add the name of the file or module in which the line occurs in parentheses. The syntax is:

```
number(filename)
```

Statement within a line

To refer to a statement within a line, use the line number followed by the number of the statement within the line, in the form:

```
line-number.statement-number
```

So, for example, 100.3 refers to the third statement in line 100.

7.5.3 Expressions

Some debugger commands require expressions as arguments. Their syntax is based on C. A full set of operators is available. The lower the number, the higher the precedence of the operator. In descending order of precedence these are:

Precedence	Operator	Purpose	Syntax
1	`()`	Grouping	`a * (b + c)`
	`[]`	Subscript	`isprime[n]`
	`.`	Record selection	`rec.field,a.b.c`
	`->`	Indirect selection (in fact 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`

Table 7-1: armsd expressions

Precedence	Operator	Purpose	Syntax
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`

Table 7-1: armsd expressions (Continued)

Subscripting can only be applied to pointers and array names. armsd will check both the number of subscripts and their bounds in languages which support such checking. Out-of-bound array accesses will be warned against. 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` will yield 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.

7.5.4 Constants

Constants may be decimal integers, floating-point numbers, octal integers or hexadecimal integers. Note that `1` is an integer whereas `1.` is a floating-point number.

Character constants are also allowed. For example `'A'` yields 65, the ASCII code for ‘A’.

Address constants may be specified by the address preceded with an ‘@’ symbol. For commands which accept low-level symbols by default, the ‘@’ may be omitted.

7.5.5 Names used in syntax descriptions

These terms are used in the following sections for the command syntax descriptions.

`context`	is the program’s activation state. See ➲*7.5.1 Variable names and context* on page 7-6.
`expression`	is an arbitrary expression using constants, variables and the operators described in ➲*7.5.3 Expressions* on page 7-8. It is either a low-level or a high-level expression, depending on the command. `list`, `find`, `examine`, `putfile`, and `getfile`, require low-level expressions as arguments; all others require high-level expressions. Low-level expressions are arbitrary expressions using constants, low-level symbols and operators. High-level variables may be included in low-level expressions if their specification starts with # or $, or if they are preceded by ^. High-level expressions are arbitrary expressions using constants, variables and operators. Low-level symbols may be included in high-level expressions by preceding them with @.
`location`	is a location within the program. (➲*7.5.2 Program locations* on page 7-7)
`variable`	is 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 ➲*7.5.1 Variable names and context* on page 7-6 to see the variable elsewhere in the program.
`format`	is either: • *`hex`* • *`ascii`* • *`string`* (This is a sequence of characters enclosed in double quotes ("). A backslash (\) may be used as an escape character within a string. • a C `printf` function format descriptor. ➲*Table 7-2: Format descriptors* on page 7-11 shows some common descriptors.

Type	Format	Description
int		Only use this if f the expression being printed yields an integer
	%d	Signed decimal integer (default for integers)
	%u	Unsigned integer
	%x	Hexadecimal (lower case letters) - same as hex
char		Only use this if the expression being printed yields an integer
	%c	Character-same as ascii
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 is safe with any kind of pointer
float		Only use this for floating-point results
	%e	Exponent notation, eg. 9.999999e+00
	%f	Fixed point notation, eg. 9.999999
	%g	General floating point notation, eg. 1.1, 1.2e+06

Table 7-2: Format descriptors

7.6 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
```

will print 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 is `%g`. Pointer values are treated as integers, using a default fixed format `%.8x`, for example, 000100e4.

See *Watch* in ❯*7.9 Controlling Execution* on page 7-17 for details of possible difficulties with register variables.

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 will be treated as a word address, and memory at that location (and if necessary the following locations) will be changed to the values in the following expression(s).

7.6.1 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:

`$clock`	number of microseconds since simulation started.
`$cmdline`	argument string for the debuggee
`$echo`	non zero if commands from obeyed files should be echoed (initially set to 0).
`$examine_lines`	default number of lines for examine command (initially set to 8).
`$format`	default format for printing integer values (initially set to "%d").
`$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.
`$fr_full`	non zero if the `fpregisters` command should print exact contents (initially set to 0).
`$inputbase`	base for input of integer constants (initially set to 10).
`$list_lines`	default number of lines for list command (initially set to 16).
`$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`	this variable can be used to output any statistics which the ARMulator has been keeping. This variable is read-only.
`$statistics_inc`	this variable is similar to `$statistics`, but outputs the difference between the current statistics and those when `$statistics` was last read. This variable is read only.
`$type_lines`	default number of lines for the type command.
`$vector_catch`	indicates whether or not execution should be caught when various conditions arise. The default value is %RUsPDAifE. Capital letters indicate that the condition is to be intercepted.

R	reset
U	undefined instruction
S	SWI
P	prefetch abort
D	data abort
A	address exception
I	IRQ
F	FIQ
E	Error

7.6.2 Formatting integer results

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 ➲*7.5.5 Names used in syntax descriptions* on page 7-10.

7.6.3 Specifying the base for input of integer constants

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 recognised 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 setting.

7.7 Symbols

This command lists all symbols (variables) defined in the given or current context, along with their type information.

```
    sy{mbols} {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.

`symbols $` gives internal variables.

7.7.1 Variable

The `variable` command provides type and context information on the specified variable (or structure field).

```
    v{ariable} variable
```

`variable` can also return the type of an expression.

7.7.2 Arguments

This command is used to show the arguments that were passed to the current procedure, or another active procedure. :

```
    a{rguments} {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.

7.8 Accessing and Executing Programs

Go

This command 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. The syntax is:

```
g{o} {w{hile} expression}
```

If `while` is used, *`expression`* is evaluated whenever a breakpoint is reached. if *`expression`* evaluates to true (ie. non-zero) the breakpoint is not reported and execution continues.

Getfile

This command reads the contents of an area of memory from a file. Its syntax is:

```
ge{tfile} filename expression
```

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.

Load

This command loads an image for debugging. Its syntax is:

```
lo{ad}{/callagraph} image-file {arguments}
```

`image-file` is the name of the program to be debugged, and *`arguments`* are the command-line arguments the program would normally take when run. `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, reload, setting of `$cmdline`, or command-line invocation are used again. The `load` command clears all breakpoints and watchpoints.

If specified, `/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).

Sourcedir

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`.

Putfile

This command writes the contents of an area of memory to a file. Its syntax is:

```
pu{tfile} 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 '+';
- the value of *expression1* + *expression2* - 1 if *expression2* is preceded by '+'.

The file is written as a sequence of bytes. Low level symbols are accepted by default.

Reload

This command reloads the object file specified on the armsd command line, or the last load command. Its syntax is:

```
rel{oad} {arguments}
```

If no arguments are specified, the arguments used in the most recent load, reload, setting of `$cmdline` or command line invocation are used again. Breakpoints (but not watchpoints) remain set after a `reload` command.

Type

This command types the contents of a source file, or any text file, between a specified pair of line numbers. Its syntax is:

```
t{ype} {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 *expressions2* 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.

7.9 Controlling Execution

Break

You can specify breakpoints at:

- procedure entry and exit
- lines
- statements within a line

The syntax of the `break` command is:

```
b{reak}{/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
`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`* should 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`	using this `if` clause 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.

Deleting breakpoints: Use `unbreak` to delete any unwanted breakpoints, referring to them by their number preceded by #. All breakpoints can also be deleted by referring to them by location.

Call

This command calls a procedure. The syntax is:

```
ca{ll}{/size} location {(expression-list)}
```

`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,you can examine it using:
	`print $result` for integer variables
	`print $fpresult` for floating point variables.

Istep

This command steps execution through one or more instructions. Its syntax is:

```
is{tep} {in} {count|whi{le} expression}
is{tep} out
```

Its behaviour is identical to that of the `step` command when the `language` has been set to `none`.

Return

This command returns to the caller of the current procedure, passing back a result where required. The syntax of this command is:

```
ret{urn} {expression}
```

Note: There is no way to 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 or expression or compound type.

Step

This command steps execution through one or more statements. The syntax is:

```
s{tep} {in} {out} {count|whi{le} expression}
s{tep} 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 (ie. zero).
`out`	Steps out of a function to the line of originating code which immediately follows that function. This option is useful when `step in` has been used too often.

To step by instructions rather than statements, set the language to `none` (using `language none`), or use the `istep` command.

Unbreak

This command removes a breakpoint. Its syntax is:

```
unb{reak} {location}
```

`location` is either a source code location, or # followed by the breakpoint number, as displayed by `break`.

If there is only one breakpoint, deleted it using `unbreak` without any arguments .

Note: Breakpoints are not renumbered following deletion of other breakpoints unless the breakpoint deleted was the last one set. Once a breakpoint has been assigned a number it keeps it.

Unwatch

This command clears a watchpoint. Its syntax is:

```
unw{atch} {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`.

Watch

This command is used to set a watchpoint on a variable. When the variable is altered, program execution is suspended.

The syntax is:

```
w{atch} {variable}
```

If `variable` is not specified, a list of current watchpoints is displayed along with their numbers. As with `break/unbreak`, these numbers can subsequently be used to remove watchpoints.

Bitfields are not watchable.

Notes: The existence of 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 probably more practical to set a breakpoint in the area of suspicion and set watchpoints once execution has been stopped there.

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.

7.9.1 armsd watchpoints and breakpoints and EmbeddedICE

`armsd` provides `break`, `watch`, `unbreak` and `unwatch` commands. Their behaviour is unchanged when used with EmbeddedICE. However, you should note the following points:

armsd watchpoints

All armsd watchpoints are data changed watchpoints, that is, they are not activated if the data is read or written to, with the same data currently in memory.

You can implement other forms of watchpoints by accessing EmbeddedICE macrocell registers directly (see *18.4 Accessing the EmbeddedICE Macrocell Directly* on page 18-3).

Inspecting points

When inspecting breakpoints and watchpoints using the `break` and `watch` commands with no arguments, the output specifies whether they are hardware or software points.

Hardware vs software breakpoints

Hardware breakpoints are implemented using an EmbeddedICE macrocell point to spot an instruction fetch from the appropriate address. This works in all cases, even if the program being debugged modifies itself as it executes, or if the code is in ROM. However, it completely ties up one EmbeddedICE macrocell point.

Software breakpoints are implemented using an EmbeddedICE macrocell point to spot an instruction fetch of a particular bit pattern. This bit pattern will have previously been stored at the appropriate location, and the instruction noted. Therefore, self-modifying code or code in ROM cannot be debugged using this type of breakpoint (EmbeddedICE will not attempt to use software breakpoints for code in ROM). Any number of software breakpoints can be supported using a single EmbeddedICE macrocell point.

Hardware watchpoints are implemented using an EmbeddedICE macrocell point to spot data writes to addresses which fall inside a mask. This type of watchpoint is efficient, as execution will only stop when the relevant data area is written, but ties up an EmbeddedICE macrocell point completely. Note also that if a structure or array is being watchpointed, the mask is likely to include some addresses which are not part of the object being watchpointed. In this case writes to these unwanted addresses will be filtered out by EmbeddedICE, but execution performance will be slightly degraded.

Software watchpoints make no use of EmbeddedICE macrocell. Instead after each instruction, the data concerned is read after each instruction and if it has changed execution is halted, otherwise execution is resumed. This type of watchpoint reduces execution performance drastically. In addition, it clearly cannot be used on write-only areas of memory such as some memory mapped device registers.

The program counter after a watchpoint

Watchpoints are taken when the data being watchpointed has changed. Thus, when a watchpoint is taken, the PC will point to the instruction after the one which caused the watchpoint to be taken, and the value of the watchpointed data will be the new value, not the old value.

7.10 Program Context

Where

This command prints the current context as a procedure name, line number in the file, filename and the line of code. Its syntax is:

```
wh{ere} {context}
```

If a context is specified after the `where` command, the debugger will display the location of that context.

Backtrace

This command prints information about all currently active procedures, starting with the most recent, or for a given number of levels, specified using `count`. The syntax is:

```
ba{cktrace} {count}
```

Context

This is used to set the context in which the variable lookups will occur. 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. The syntax of this command is:

```
con{text} context
```

If *`context`* is not specified, the context will be reset to the active procedure.

Out and In

These commands are a shorthand way of changing the current context by one activation level. Their syntax is:

`ou{t}`	sets the context to that of the caller of the current context
`in`	sets the context to that called from the current level.

It is an error to issue an `out` or `in` command when no further movement in that direction is possible.

7.11 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.

7.11.1 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 initialisation, 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 ‘^’.

Predefined low-level symbols

There are several predefined low-level symbols:

`r0` to `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.
`psr cpsr spsr`	`psr` and `cpsr` are synonyms for the current mode’s processor status register. `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 Application Note 11, *Differences Between ARM6 Series and Earlier Processors* for more infomation.

`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 may allocate as it pleases (stored in r4 to r10).
`sb`	Static base, as used in reentrant 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).
`fp`	The frame pointer (r11).
`ip`	Used in procedure entry and exit and as a scratch register (r12).
`sp`	The stack pointer (r13).
`lr`	The link register (r14).

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).

`let` can also set the processor status register (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.

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
```

7.11.2 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 behaviour 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 the `language` command to set this up. The syntax is:

```
la{nguage} {none|C|F77|PASCAL|ASM}
```

7.11.3 Registers

This command displays the contents of ARM registers 0 to 14, the program counter (PC) and the status flags contained in the processor status register (PSR). The syntax is:

`r{egisters} {`*`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. In addition to the mode names listed in *7.11.1 Low-level symbols* on page 7-23, mode may take the value `all`, in which case the contents of all registers of the current mode are displayed, together with all banked registers for other modes with the same address width.

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
```

7.11.4 Fpregisters

This command displays the contents of the eight floating point registers f0 to f7 and the floating point processor status register FPSR. Its syntax is:

```
    f{pregisters}
```

There are two formats for the display of floating point registers, selected using the root-level variable `$fr_full` with `let`.

The simpler form displays the registers and FPSR, and the full version includes detailed information on the floating point numbers in the registers. Use:

```
    {let} $fr_full = 0
```

to produce the following display:

```
f0 = 0           f1 = 3.1415926535 f2 = Inf          f3 = 0
f4 = 3.1415926535 f5 = 1           f6 = 0            f7 = 0
fpsr = %IZOux_izoux
```

Use the alternative:

```
{let} $fr_full = 1
```

to produce the more detailed display:

```
f0 = 0           f1 = 3.1415926535 f2 = Inf          f3 = 0
f4 = 3.1415926535 f5 = 1           f6 = 0            f7 = 0
fpsr = %IZOux_izoux
f0 = I + 0x3fff 1 0x0000000000000000f1 = I + 0x4000 1 0x490fdaa208ba2000
f2 = I +u0x43ff 1 0x0000000000000000f3 = I - 0x0000 0 0x0000000000000000
f4 = I + 0x4000 1 0x490fdaa208ba2000f5 = I + 0x3fff 1 0x0000000000000000
f6 = I + 0x0000 0 0x0000000000000000f7 = I + 0x0000 0 0x0000000000000000
fpsr = 0x01070000
```

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 for single, D for double, E for extended, I for internal format, and P for 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 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; upper case means the flag is set and lower case means that it is cleared. The flags are:

`I`	Invalid operation
`Z`	Divide by zero
`O`	Overflow
`U`	Underflow
`X`	Inexact

7.11.5 Examine

This command allows you to examine the contents of the memory between a pair of addresses, displaying it in both hexadecimal and ASCII formats, with 16 bytes per line. Its syntax is:

```
e{xamine} {expression1} {, {+}expression2 }
```

Low level symbols are accepted by default.

Start address

The start address is given by *`expression1`*. If this is omitted 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.

End address

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).

7.11.6 List

This command displays contents of the memory between specified pair of addresses in hexadecimal, ASCII and instruction format, with 4 bytes (one instruction) per line. The syntax is:

```
l{ist}{/size} {expression1}{, {+}expression2 }
```

Low-level symbols are accepted by default.

You can follow *`list`* by an optional `size` specifier:

`/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. This usually chooses the correct size, but you can set the size explicitly.

Start address

`expression1` gives the start address. If no address is specified, the default setting is either:

- the address associated with the current context minus 32, if the context has changed since the last list command was issued;
- the address following the last address displayed by the last `list` command, if the context has not changed since the last `list` command was issued.

End address

The end address is given by *`expression2`*. It and 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).

7.11.7 Find

This command finds all occurrences in memory of a given integer value or character string. Its syntax is either of the following:

```
fi{nd} expression1 {,expression2 {,expression3}}
fi{nd} 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`.

7.11.8 Lsym

This command displays low-level symbols and their values. Its syntax is:

```
ls{ym} pattern
```

`pattern` is a symbol name or part of a symbol name.

Wildcards: A wildcard (indicated by *) can be used at the beginning and/or end of the pattern to match any number of characters:

`ls *fred`	displays information about fred, alfred
`ls fred*`	displays information about fred, frederick
`ls *fred*`	displays information about alfred, alfreda, fred, frederick

The wildcard ? matches one character:

`ls ??fred`	matches Alfred
`ls Jo?`	matches Joe, Joy, and Jon

7.12 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 writeable) by a single coprocessor data transfer (CPDT) or a coprocessor registers transfer (CPRT) instruction in a non-user mode. For coprocessors with more exotic registers, there must be support code in a debug monitor.

7.12.1 Coproc

This command describes the register set of a coprocessor and specifies how the contents of the registers are formatted for display. The syntax is:

```
c{oproc} cpnum {regdesc}*
```

`regdesc` may describe one register, or a range of registers which are accessed and are to be formatted uniformly. It has the syntax:

```
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 writeable
`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 whether the register is to be accessed through CPRT instructions.

If so, it comprises four integer values separated by space or comma:

```
r0_7, r16_23, w0_7, w16_23
```

to 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.

If not, it comprises two integer values:

```
b12_15, b22
```

to form bits 12 to 15 and bit 22 of CPDT instructions to read and write the register.

`displaydesc` is one of the items listed in *Table 7-3: Displaying the contents of coprocessor registers* on page 7-31.

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 on 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 on 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 upper case if the corresponding bit of the field is set, in lower case otherwise. Otherwise, the lowest bit of the field corresponds to the first letter if *`bit1`* < *`bit2`*, to the last letter.	

Table 7-3: Displaying the contents of coprocessor registers

For example, the floating-point coprocessor might be described by the command (reformatted for clarity):

```
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
```

7.12.2 Cregisters

This command displays the contents of all readable registers of a coprocessor, in the format specified by an earlier `coproc` command. The syntax is:

```
cr{egisters} cpnum
```

7.12.3 Cwrite

This command writes to a coprocessor register. The syntax is:

```
cw{rite} cpnum rno val {val}*
```

Register *rno* of coprocessor *cpnum* must have been specified as writeable; each *val* is an integer value and there must be one *val* item for each word of the coprocessor register.

7.13 Miscellaneous Commands

7.13.1 Alias

This command defines, undefines or lists aliases. It allows you to define your own symbolic debugger commands:

```
al{ias} {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 ('``'):

7.13.2 Comment

This command writes a message to `stderr`:

```
com{ment} message
```

7.13.3 Help

This command displays a list of available commands, or help on a particular command:

```
h{elp} {command}
```

If information about all commands as well as their names is required, type `help *`. The help displayed includes syntax and a brief description of the purpose of each command.

7.13.4 Log

This command sens the output of subsequent commands tt 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 input/output to or from the program being debugged is not logged.*

7.13.5 Obey

This command executes a set of debugger commands which have previously been stored in a file, as if they were being typed at the keyboard:

```
o{bey} command-file
```

where *`command-file`* is the name of the file containing the list of commands to be executed.

7.13.6 Pause

This command prompts the user to press a key to continue:

```
pa{use} prompt-string
```

The prompt string is written to stderr, and execution continues only when a key is pressed. If ESC is pressed and commands are being read from a file, the file is closed before execution continues.

7.13.7 ProfClear

This command resets profiling counts:

```
profc{lear}
```

7.13.8 ProfOn

This command starts collecting profiling data:

```
pro{fon} {interval}
```

`interval` is the time between PC-sampling in microseconds. Lower values have a higher performance overhead, and will slow down execution, but higher values are not as accurate.

7.13.9 ProfOff

This command stops collectin profiling data:

```
profof{f}
```

7.13.10 ProfWrite

This command writes profiling information to a file:

```
profw{rite} {filename}
```

The generated information can be viewed using the `armprof` utility, as described in ➲*Chapter 8, ARM Profiler*.

7.13.11 While

This command is only valid at the end of an existing statement. You enter multi-statement lines by separating the statements with ';' characters:

```
statement; {statement;} whi{le} expression
```

Interpretation of the line continues until *`expression`* evaluates to false (zero).

7.13.12 Quit

This command terminates the current symbolic debugger session and closes any open log or obey files:

```
q{uit}
```

7.13.13 !

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.

7.13.14 | (vertical bar)

This introduces a comment line.

7.13.15 ; (semicolon)

This is not a command, but may be used to separate 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 will not be executed until all previously queued commands have been executed.

7.14 Automatic Command Execution on Startup

The symbolic debugger will obey commands from an initialisation file if one exists before it reads commands from the standard input. The initialisation file is called `armsd.ini`:

The current directory is searched first, and then the directory specified by the environment variable `HOME`.

7.15 Performance simulation using armsd

You can simulate the performance of an ARM/Thumb system consisting of an ARM/Thumb processor attached to one or more sections of internal or external memory.

Before performing the simulation you must define:

- The speed at which the ARM/Thumb processor is clocked.
- The number of regions of memory attached and, for each region:
  - the address range to which that region is mapped
  - the databus width (8, 16 or 32 bits)
  - the access times for the memory region

You must then create a file called `armsd.map` which describes your memory layout. This file consists of a number of lines, each line describing one region of memory.

7.15.1 Format of the armsd.map file

The format of each line is as follows:

```
start size name width access read-times write-times
```

The fields are:

start	is the start address of the region of memory in hexadecimal.
size	is the size of the region of memory in hexadecimal.
name	is a single word which can be used to identify the region of memory when the memory access statistics are displayed. This name is of no significance to armsd, so any name can be used, but to ease readability of the memory access statistics, use a descriptive name such as SRAM, DRAM, EPROM
width	is the width of the databus in bytes (ie. 1 for an 8 bit bus, 2 for a 16-bit bus or 4 for a 32-bit bus).
access	describes the type of access which may be performed on this region of memory. The `r` is for read-only, `w` for write-only, `rw` for read-write, or `-` for no access. The character '*' may be appended to the access to describe a system which uses a 32-bit databus but which has a 16 bit latch to latch the upper 16 bits of data so that a subsequent 16-bit sequential access may be fetched directly out of the latch.

read-times	describes the non-sequential and sequential read times in nanoseconds. These should be entered as the non-sequential read access time followed by / (slash), followed by the sequential read access time. Omitting the / and using only one figure indicates that the non-sequential and sequential access times are the same.
write-times	describes the non-sequential and sequential write times. The format is identical to that of read times.

Examples are given below:

```
0 80000000 RAM 4 rw 135/85 135/85
```

This describes a system with a single contiguous section of RAM from `0` to `7FFFFFFF` with a 32-bit databus, read/write access and N and S access times of 135 and 85 nanoseconds respectively.

This is typical of a 20MHz PIE (Platform Independent Evaluation) card. Note that the N cycle access time is one clock cycle longer than the S cycle access time. For a faster system a smaller N cycle access time should be used, for example: for a 33MHz system the access times would be defined as 115/85 115/85.

```
0 80000000 RAM 1 rw 120/70 120/70
```

This describes a system with the same single contiguous section of memory, but with an 8-bit external databus and slightly faster access times.

You should not round the access times up to the nearest clock cycle. The symbolic debugger will do this in any case, however, in the case where an 8-bit or 16-bit databus is attached the debugger will need the precise access times to calculate the overall access time for a 32-bit access, ie. A sequential 32-bit access on the above system would take: 4 * 70 = 280nS

This is then rounded up to the next cycle (to 300nS with a 20MHz clock). If the figures given for the access times were rounded up to the nearest clock cycle (ie. 150/100 in this case) a 32-bit access would take 4 * 100 = 400nS

Note: *It may be the case that external memory accesses should be rounded up to the next clock cycle, if, for example, the memory system uses the same clock when latching the 4 bytes of data to perform the 32-bit access.*

```
>00000000 8000     SRAM  4 rw 1/1 1/1
>00008000 8000     ROM   2 r  70/70 70/70
>00010000 8000     DRAM  2 rw 135/85 135/85
>7fff8000 8000     Stack 2 rw 135/85 135/85
```

This describes a system with four regions of memory:

- A fast region of memory from `0` to `7FFF` with a 32-bit databus.
- A slower section of memory from `8000` to `FFFF` with a 16-bit databus. This is labelled ROM and will contain the image code, hence it is marked as read-only.

- Two sections of RAM, one from `10000` to `17FFF` which will be used for image data and one from `7FFF8000` to `7FFFFFFF` which will be used for stack data (The stack pointer is initialised to `0x80000000`).

This would be typical of an embedded system with 32K on-chip memory, 32K external 16-bit ROM and 32K external DRAM which will be used for both the image data and the stack data. In the description above this is described as two regions of memory, however in the final hardware these two would be combined. This does not make any difference as far as the simulation is concerned.

Note that the SRAM region is given access times of 1nS. In effect this means that each access will take 1 clock cycle as armsd rounds this up to the nearest clock cycle, however, specifying it as 1nS allows the same map file to be used for a number of simulations with differing clock speeds without having to update the map file.

Note: *To ensure accurate simulations you should take care when specifying the memory map that all areas of memory which the image you are simulating is likely to access are described in the memory map.*

To ensure that you have described all areas of memory you think the image should access, you can define a single memory region which covers the entire address range as the last line in the `armsd.map` file.

For example to the above description you could add a line:

```
00000000 80000000 Dummy 4 - 1/1 1/1
```

You can then detect if any reads or writes are occurring outside the regions of memory you expect using the `print $memory_statistics` command described in ➲*7.15.4 Reading the memory statistics* on page 7-38. This can prove a very useful debugging tool.

7.15.2 Specifying the clock speed

Before performing a simulation you must specify the clock speed of the processor. This may be done either by using reconfig to change the configured clock speed of the symbolic debugger or by specifying the clock speed on the debugger command line.

For details on how to change the configured clock speed of the symbolic debugger see ➲*Chapter 12, ARM Tool Reconfiguration Utility*.

To specify the clock speed on the command line use the `-clock` option. This should be followed by the clock speed in Hz. If you wish to specify the clock speed in MHz place MHz after the clock speed, for example:

```
armsd -clock 33000000 test_prog
armsd -clock 33MHz test_prog
```

7.15.3 Reading the simulated time

When performing a simulation, the symbolic debugger keeps track of the total time elapsed. This value may be read either by the simulated program or from the armsd command line.

Reading the simulated time from assembler

To read the simulated clock from a program use SWI 0x61 (SWI_Clock). This is described in ➲*17.3 Standard Monitor SWIs* on page 17-4.

Reading the simulated time from C

The standard C library function `clock()` which returns the number of elapsed centi-seconds calls SWI 0x61 so this may be used directly from C programs.

Reading the simulated time from armsd

The variable `$clock` contains the number of microseconds since simulation started. To display this value, use the command:

```
Print $clock
```

Reducing the time required for simulation

You may be able to significantly reduce the time taken for a simulation by dividing the specified clock speed by a factor of 10 and multiplying the memory access times by the corresponding factor of 10. Take the time reported by the clock() function (or by SWI_Clock) and divide by the same factor of 10.

The reason this works is because the simulated time is recorded internally in nanoseconds but SWI_Clock only returns centiseconds. Therefore dividing the clock speed by 10 shifts digits from the nanosecond count into the centisecond count allowing the same level of accuracy but taking only 1/10th the time to simulate.

7.15.4 Reading the memory statistics

To read the memory statistics use the command:

```
Print $memory_statistics
```

The statistics will be reported in the following form.

```
address   name w accR(N/S) W(N/S) reads(N/S)    writes(N/S)  time (ns)

00000000 Dummy 4 -  1/1    1/1    0/0           0/0          0
7FFF8000 Stack 4 rw 135/85 135/85 285237/82906  82906/145629 83321400
00008000 RO    4 r  70/70  70/70  1248865/38069713806971/90290578890740
00000000 SRAM  4 rw 135/85 135/85 27/0          0/0          4050
```

You can use `Print $memstats` as a shorthand version of `Print $memory_statistics`.

7.15.5 Dhrystone simulation example

1 Compile the Dhrystone program in the `examples` directory using one of the following commands:

   ARM: `armcc -o dhry_32 -Otime -DMSC_CLOCK dhry_1.c dhry_2.c`
   Thumb: `tcc -o dhry_16 -Otime -DMSC_CLOCK dhry_1.c dhry_2.c`

2 Create the following `armsd.map` file:

```
00000000 80000000 RAM 4 rw 135/85 135/85
```

3 Run the benchmark with the command:

```
armsd -clock 20MHz dhry_32
```

4 Enter `go` to start execution.

5 When requested for the number of Dhrystones, enter 35000. The program will report the number of Dhrystones per second.

6 Record the value and repeat the simulation with the Thumb version of Dhrystone (dhry_16).

You may get slightly different figures depending on the version of compiler and library you are using. Try varying the clock speed, the memory access speeds and the databus width to see the effect of these on performance.

When measuring Thumb on a 32-bit memory system, try placing a `*` after the memory access `rw` (ie. enter `rw*`) to see the performance gain from putting a 16-bit latch on such a system. The following tables show some sample results. Your results may vary depending on compiler version, compiler options and the library version.

Note: *These results are for Dhrystone version 2.1.*

Memory	ARM	Thumb
32-bit memory	14204.5	11876.5
32-bit memory	14204.5	13636.4 (with 16-bit latch)
16-bit memory	7894.7	10067.1
8-bit memory	4731.9	5703.4

Table 7-4: Clock speed = 20MHz, Memory access times (N = 135nS, S = 85nS)

Memory	ARM	Thumb
32-bit memory	16759.8	14018.7
32-bit memory	16759.8	17142.9 (with 16-bit latch)
16-bit memory	9063.4	11718.7
8-bit memory	4724.4	6237.0

Table 7-5: Clock speed = 33MHz, Memory access times (N = 115nS, S = 85nS)

Memory	ARM	Thumb
32-bit memory	52083.3	43478.3
32-bit memory	52083.3	43478.3 (with 16 bit latch)
16-bit memory	27624.3	35971.2
8-bit memory	14285.7	18939.4

Table 7-6: Clock speed = 33MHz, Memory access times (N = 30nS, S = 30nS)

7.16 Semihosting under EmbeddedICE

Two new `armsd` internal variables have been added for semihosting support. These are only supported by EmbeddedICE:

`$semihosting_enabled`	Enable semihosting
`$semihosting_vector`	Set up semihosting SWI vector

These should be set up prior to starting execution of an image, and should not be changed further as execution progresses. In general, the following settings are used:

```
$semihosting_enabled          = 1
$semihosting_vector           = 8
```

The values above are the default values.

However, if semihosting is enabled (`$semihosting_enabled = 1`), a breakpoint is set up on the SWI vector and EmbeddedICE checks to see whether the SWI being requested is one of the ARM Debug Monitor SWI's (see the Technical Specifications for full details), or a debuggee-specific SWI. If it is a Debug Monitor SWI, EmbeddedICE will emulate it, and restart execution transparently. If it is not an ARM Debug Monitor SWI execution will be restarted by executing the normal SWI code (providing the vector instruction is a branch or an LDR PC, [PC, #n]).

If there were frequently called SWIs which were not ARM Debug Monitor SWIs this method of semihosting would prove inefficient, as execution would stop much more frequently than was actually required.

A refinement upon this allows you to specify a Debug Monitor SWI Vector. Do this by setting `$semihosting_vector` to the address of this vector. When set,the semihosting SWI vector specified is breakpointed instead of the standard SWI vector. The intention is that the debuggee software's SWI code should check for all non Debug Monitor SWI's and process them, and then for Debug Monitor SWIs branch to the semihosting SWI vector. This method will ensure that SWIs only cause a breakpoint to be taken when it really needs to be.

Note that when `$semihosting_enabled` has been set to 1, the default value for `$semihosting_vector` is 0. In this state the normal SWI vector is used, and all exceptions and interrupts are trapped and reported as an error condition, no matter what the value of `$vector_catch`. If however, `$semihosting_vector` is set to 8, the normal SWI vector is still used, but exceptions and interrupts are not considered errors, and are only trapped and reported if `$vector_catch` indicates that they should be.

8 ARM Profiler

This chapter describes the ARM Profiler.

ARM Profiler

8.1 About armprof

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). In this release, profiling is available only for programs loaded into store by the debugger; function call counting will never be 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 focussed (omitting initialisation code, for example). However, care should be taken that the time between turning sampling on and off is long compared with the sample interval, or the data generated may be meaningless. Note also that turning sampling on and off does not affect the gathering of call counts.

8.2 Command-line Options

A number of options are available to control the format and amount of detail present in the profiler output.

`-Parent`	tells the profiler to display 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`	tells the profiler to display 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`	turn off the child listing.
`-Sort Cumulative`	tells the profiler to sort the output by the total time spent in each function and all of its children.

`-Sort Self`	tells the profiler to sort the output by the time spent in each function (excluding the time spent in its children).
`-Sort Descendants`	tells the profiler to sort the output by the time spent in all of a function's children but excluding time spent in the function itself.
`-Sort Calls`	tells the profiler to sort 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.

8.2.1 Example

```
armprof -parent sort.prf
```

8.3 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 (ie. 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.

The `cum%` column 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 the `self%` column gives the percentage time spent in this function itself, the `desc%` column gives the percentage time spent in the children of this function, and the `calls` column gives the number of calls to this function.

For a parent function the `self%` column gives the percentage time spent in the current function itself on behalf of this parent, the `desc%` column gives the percentage time spent in the children of the current function of behalf of this parent and the `calls` column gives the number of calls made by this parent to the current function.

For a child function the `self%` column gives the percentage time spent in this child on behalf of the current function, the `desc%` column gives the percentage time spent in this child's children on behalf of the current function and the `calls` column gives the number of times this child was called by the current function.

9 ARM Librarian

This chapter describes the ARM librarian tool.

ARM Librarian

9.1 About armlib

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 due to 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 ➲*6.4 Area Placement and Sorting Rules* on page 6-11.

The full specification of ARM Object Library Format can be found in chapter ➲*21.3 ARM Object Library Format* on page 21-26.

9.2 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`	give on-line details of the armlib command
`-c`	create a new library containing files in *`file-list`*
`-i`	insert files in *`file-list`* into the library. Existing members of the library are replaced by members of the same name.
`-d`	delete members in *`member-list`*
`-e`	extract members in *`member-list`*, placing them in files of the same name
`-o`	add an external symbol table to an object library
`-l`	list library. This may be specified together with any other option.
`-s`	list symbol table. This may be specified together with any other option.
`-v` *`file`*	Additional arguments are read in from a via file, in the same way as the armlink *`-via`* option. See ➲*6.2.1 General options* on page 6-3

10 ARM Object Format Decoder

This chapter describes the ARM Object Format Decoder tool.

ARM Object Format Decoder

10.1 About decaof

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 ➲*21.2 ARM Object Format* on page 21-10.

10.2 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 on-line 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
```

11 ANSI to PCC C Translator

This chapter describes the ANSI C to PCC C translator.

ANSI to PCC C Translator

11.1 About topcc

The program topcc helps to translate (suitable) C programs and headers from the ANSI dialect of C into the PCC dialect of C, primarily by re-writing top-level function prototypes (whether declarations or definitions).

topcc 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 in which function prototypes have been obscured by, for example, preprocessor macros.

The translation performed is limited, and other differences between the ANSI and PCC dialects must be dealt with in the source after or (preferably) before translation.

11.2 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`	don't remove keyword `const`.
`-e`	don't remove `#error....`
`-p`	don't remove `#pragma....`
`-s`	don't remove keyword `signed`.
`-t`	don't remove 2nd argument to `va_start()`.
`-v`	don't remove keyword `volatile`.
`-l`	don't add `#line` directives.

11.3 Translation Details

Primarily, topcc rewrites top-level function protoypes, whether definitions or declarations.

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 re-written the PCC way. 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 '..." in a function definition is replaced by `int va_alist`, and 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 re-synchronise line numbering. These quote the source filename, so that debugging tools then refer to the ANSI form of sources.

11.4 Issues with topcc

topcc takes no account of the setting of conditional compilation options. This is quite deliberate: it converts all conditionally compilable variants in parallel.

A price to be paid is that braces must be nested reasonably within conditionally compilable sections, or topcc may lose track of the brace nesting depth, which is used to determine whether it is within, or between top-level definitions and declarations.

In principle, tracking brace-nesting depth oblivious of preprocessing is impossible. In practice, topcc uses heuristics to match conditionally compiled braces, usually successfully. If topcc finds that it is lost it complains of “mis-matched, conditionally included braces”.

A second, niggling restriction is that 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–but then these can just as easily be eliminated from the input program by the user.

The one disaster for topcc is to find an extra closing brace and to start processing text prematurely as if it were at the top level. This leads to damage to function calls and macro invocations. In general it is a good idea to compare the output of topcc with its input (using a file difference utility), as a check that changes have been reasonably localised to function headers and declarations. If necessary, most of topcc’s other transliterations can be inhibited to make these principal changes more visible (See *11.2 Command Line Options* on page 11-2*).*

12 ARM Tool Reconfiguration Utility

This chapter describes the utility used to reconfigure the default settings of the ARM tools.

ARM Tool Reconfiguration Utility

12.1 About reconfig

This utility allows you to reconfigure the default settings of the ARM tools. It is useful to do this when their current settings do not reflect the system in which you are using them. Settings include Serial line Baud Rates for the ARM Symbolic Debugger and the size (in bits) of the Program Counter for code produced by the ARM C Compiler.

`reconfig` can be run in a command-line mode, silently, or as a full-screen option editor.

When driven as a full-screen option editor, the program uses ANSI escape sequences to drive the screen. Consequently, `reconfig` cannot be used in this mode on HP-UX when started up in an HPTERM.

The tool settings are stored in the executable file for each tool. This utility patches in any changes directly.

12.2 Tool Reconfiguration

The `reconfig` utility can configure all the major ARM tools. Their options are described in the following sections.

12.2.1 Thumb C compiler (tcc)

The Thumb C compiler options and their possible values are shown below:

Option	Command-line string	Possible Values	

Bytesex	`bytesex`	host
		little
		big
			`host`
			`little`
			`big`

Stack limit check	`stacklimitcheck`	off on	`off` `on`
ARM/Thumb interworking	`interwork`	off on	`off` `on`

Table 12-1: Thumb compiler options

12.2.2 ARM C compiler (armcc)

The ARM C compiler options and their possible values are shown below:

Option	Command-line string	Possible values	

Bytesex	`bytesex`	host
		little
		big
			`host`
			`little`
			`big`

ARM/Thumb interworking	`interwork`	off on	`off` `on`

Program Counter	`pc`	26bit
		32bit
			`26bit`
			`32bit`

FP Emulation	`fpe`	fpe2 fpe3	`fpe2` `fpe3`
Stack limit checking	`stacklimitcheck`	off on	`off` `on`
Re-entrant code	`reentrantcode`	yes no	`yes` `no`
FP register args	`fpregargs`	yes no	`yes` `no`
No FP register	`nofp`	yes no	`yes` `no`
Use SW FP Library	`swfplib`	yes no	`yes` `no`
Unaligned word load/stores	`unalignedwordloadstores`	yes no	`yes` `no`
Old format asd tables	`oldformatasdtables`	yes no	`yes` `no`
Target CPU	`cpu`	arm6, arm7, arm7m, arm7tm	
Max instrs / int literal	`intloadmax`	default, 1<=user_value	
Max regs per LDM/STM	`ldmmax`	default, 3<=user_value<=16	

Table 12-2: ARM compiler options

12.2.3 ARM assembler (armasm)

The ARM assembler options and their possible values are shown below:

Option	Command-line string	Possible Values	

Bytesex	`bytesex`	host
		little
		big
			`host`
			`liitle`
			`big`

Program Counter	`pc`	26bit
		32bit
			`26bit`
			`32bit`

Stack limit check	`stacklimitcheck`	off on	`off` `on`
Width of page	`widthofpage`	default 0 <*user_value* < 255	`default`
Lines	`lines`	default 0 <*user_value* < 255	`default`

CPU	`cpu`	Generic ARM
		ARM6
		ARM7
		ARM7M
		ARM8
			`Generic ARM`
			`ARM6`
			`ARM7`
			`ARM7M`
			`ARM8`

ARM Architecture Version	`archvsn`	3
		4
			`3`
			`4`

UMULL support	`umull`	yes no	`yes` `no`

Table 12-3: ARM assembler options

12.2.4 Thumb assembler (tasm)

The Thumb assembler options and their possible values are shown below:

Option	Command-line string	Possible Values	

Bytesex	`bytesex`	host
		little
		big
			`host`
			`little`
			`big`

Stack limit check	`stacklimitcheck`	off on	`off` `on`
Width of page	`widthofpage`	default 0 <*user_value* < 255	`default`
Lines	`lines`	default 0 <*user_value* < 255	`default`

CPU	`cpu`	Generic ARM
		ARM6
		ARM7
		ARM7m
		ARM7TM
		ARM8
			`Generic ARM`
			`ARM6`
			`ARM7`
			`ARM7M`
			`ARM7TM`
			`ARM8`

ARM Architecture Version	`archvsn`	3
		4
			`3`
			`4`

UMULL support	`umull`	yes no	`yes` `no`

Table 12-4: Thumb assembler options

12.2.5 ARM symbolic debugger (armsd)

Not all of these options are always reconfigurable. It depends on the linked objects when the executable was made. The options, and their possible values, are shown below:

Option	Command-line string	Possible Values	

Bytesex	`bytesex`	little
		big
		dontcare
			`little`
			`big`
			`dontcare*`

Load FPE	`fpe`	yes no	`yes` `no`

Serial linespeed	`seriallinespeed`	9600
		19200
		38400
			`9600`
			`19200`
			`38400`

Load FPE	`fpe`	yes no	`yes` `no`
Default tool option	`defaulttool`	armul remote	`armul` `remote`
Serial port	`serialport`	default 0 <= user_value < 255	`default`
Parallel port	`parallelport`	default 0 <= user_value < 255	`default`
Default RDP driver	`defaultrdp`	serial serpar	`serial` `serpar`

Default CPU emulated	`defaultcpu`	`arm6, arm2, arm2as,`
		`arm61, arm3, arm60,`
		`arm600, arm610, arm620,`
		`arm7, arm70, arm700,`
		`arm7d, arm70d, arm7dm,`
		`arm70dm, arm7tdm`	

CPU speed	`cpuspeed`	20, 33,(suggested values) or any value in MHz	
Old expessions	`oldexpressions`	yes no	yes no

Table 12-5: Symbolic debugger options

Note: dontcare means that armsd can debug and emulate images which have the same byte order as that with which the ARMulator component of armsd is configured. When `armsd` is delivered, the default is little endian host byte order (386/DOS) but, of course, you can configure armsd with the big endian host byte order(SPARC/SunOS) if you wish. This in turn can be overridden on the armsd command line.

12.2.6 ARM linker (armlink)

The armlink options and their possible values are shown below. All of the options refer to attempts to match an unreferenced symbol with some derivative of it:

Option	Command-line string	Possible Values	
Match _sym to sym	`_s-to-s`	no yes	`no` `yes`
sym to _sym	`s-to-_s`	no yes	`no` `yes`
Mod_Sym to Mod.Sym	`m_s-to-m.s`	no yes	`no` `yes`
sym__type to sym	`s__t-to-s`	no yes	`no` `yes`
PC-rel.relocn=>code	`pcrelimpliescode`	no yes	`no` `yes`

Table 12-6: Linker options

12.3 Using reconfig

reconfig may be used in two modes:

on-screen mode	is used when no parameters are supplied on the command line
command-line mode	the options are set with command line arguments. The changes are made silently unless errors occur.

In both modes, you must change directories so that the tools you want to reconfigure are in your current directory. Run reconfig from there.

12.3.1 On-screen mode

Run reconfig by typing:

```
reconfig
```

Each tool is checked for existence and consistency. Problems that occur are displayed on the screen. See ➲*12.4 Reconfiguration Errors* on page 12-10 for more information.

1 A menu is displayed, and you can move the select bar with TAB, SPACE and the ARROW KEYS.

2 Move the select bar onto the tool you wish to reconfigure, and press ENTER.

The current settings for the selected tool are displayed, along with the list of possible settings for each option. New settings can be assigned to the options for any tools that were found and checked out OK. Use ENTER to assign the currently highlighted setting to the selected option. If a user_value is to be entered, type it in after pressing ENTER. When moving between options, the current setting is highlighted first by default.

If a tool did not check out OK, then the possible options and settings are displayed, but these cannot be changed.

3 Type q to go back to the Main Menu. The other tools can be changed in the same way.

4 When you have reconfigured all that you require, exit from the Main Menu using the menu option to quit, or just type q. You will be asked if you wish to save the current configuration. The default is n - not to save them so press y if you do wish to save the new settings, followed by RETURN. The program will tell you whether it saved the settings.

5 Press ENTER at the prompt and you're finished.

Note If you are using anything other than a PC under DOS to run reconfig, then in order to use the onscreen mode, ANSI escape sequences generated by reconfig must be interpreted correctly by the Operating System or shell in which you are running. If this is not the case, the screen layout will be wrong and you should use the command-line mode instead (see ➲*12.3.2 Command-line mode*).

12.3.2 Command-line mode

The command line syntax is:

```
reconfig toolname [ option1=value1 [ option2=value2 ] ... ]
```

The *option* and *value* separator is either whitespace or an equals sign, these may be mixed on the same line.

If only the *toolname* is given, then the current settings are displayed. Otherwise, the specified values are assigned to the specified options.

Each option and value is matched against those permitted for that tool. Only the number of characters given on the command-line are matched, so they must be unambiguous (if not, then the first match is assumed). The strings that should be typed for the options and values are given in ➲*12.2 Tool Reconfiguration* on page 12-2.

12.3.3 Examples

The following all set the armcc tool to use a 32-bit Program Counter and to refrain from checking the stack:

```
reconfig armcc pc=32bit stackcheck=no
reconfig armcc pc 32bit stackcheck no
reconfig armcc pc=32bit stackcheck no
reconfig armcc pc=32 stack=no
reconfig armcc p=3 s=n
```

The following could be used to set the armasm Program Counter to 26 bits and the listing width to 80:

```
reconfig armasm pc=26bit widthofpage=80
reconfig armasm p 26 w 80
```

12.4 Reconfiguration Errors

This section describes the errors that may be produced by reconfig, why they occur, and (where appropriate) and how to correct them.

In on-screen mode, if the output to the screen becomes garbled, it is likely that the ANSI escape sequences are not being interpreted correctly by your shell or Operating System. This should only happen on systems running anything other than DOS. If this is the case, you must change your OS/shell to one with ANSI interpretation, or use the command-line mode.

`*filename* - File not found/writeable`

This happens when a tool could not be found in the current directory, or if present, does not have write permission. You must ensure that you are in the directory containing all the tools that you wish to reconfigure, and that you have write permission for them.

`*filename* - Can’t find configuration block`

This happens when reconfig has read an executable file with an ARM tool filename, but the tool is either corrupt, or the file is not an ARM tool at all. You must ensure that the tool filename executables in the current directory are the ARM tools and not something else with the same name(s).

`filename is not ARM’s toolname`

This happens when reconfig reads an ARM tool file executable that is not consistent with its filename. (eg. executable file armcc is really armasm). You must ensure that the installed files are not renamed to other tool names. It is best to install the tools again.

`toolname - invalid tool name`

This happens in command-line mode when the toolname specified on the command-line is not an ARM tool. Check the spelling.

`function() : Corrupted file`

This will only happen if a tool executable has become corrupted. Re-install the tool(s) and try again. If the problem persists, get another copy of the Release.

`Option ’option’ cannot be reconfigured`

This happens in command-line mode when an attempt is made to change the setting of an option that is not reconfigurable under the current tool executable configuration. This should only happen when attempting to reconfigure the armsd : default tool option.

`Option ’option’: Value ’value’ is invalid`

This happens in command-line mode when the specified value is not a valid setting for the option. Check that value and option are both unambiguous and within range for user values.

`Option ’option’ is invalid`

This happens in command-line mode when the specified option is not a valid one for the selected tool. Check that the option specified is permitted and/or spelled correctly.

13 ARM make Utility

This chapter introduces the ARM make utility included with the PC release of the ARM Software Toolkit.

ARM make Utility

13.1 About armmake

armmake assists with the management of programs, documents, applications, and other complex, structured objects made from several components, that need processing, and have some consistency contraints between them. It is used most often to ease rebuilding programs from their source code, recompiling a piece of source only when it—or other code upon which it depends—has been updated.

Details of the format of armmake makefile entries are given in ➲*13.3 Makefile Format for armmake* on page 13-4. The input to armmake is a text file describing the system to be managed. The text file is usually called `makefile`, and its format is almost identical to most other make utilities.

In its simplest form, a makefile consists of a sequence of entries which describe:

- what each component of a system depends on
- the commands that must be executed to make an up-to-date version of that component.

13.2 Command-line Options

The armmake command has the following syntax:

```
armmake [options] [list-of-targets-and-macro-definitions]
```

options are as follows:

`-f` *makefile*	Read the system description from *makefile*. *makefile* defaults to `makefile` if omitted.
`-i`	Ignore return codes from commands (equivalent to .IGNORE). armmake usually stops if it encounters a bad (non-0) return code.
`-k`	On encountering a bad (non-zero) return code, don't give up, but continue with each branch of the makefile that does not depend on the failing command. This is likely to be useful if you are rebuilding a large system after making many changes and armmake is required to do as much rebuilding as possible.
`-n`	Don't execute any commands; just show on the screen what commands would be executed, and give the reason for wanting to execute each one.
`-o` *command-file*	Don't execute commands to make the target(s) up-to-date; write them to *command-file* for later execution.
`-s`	Don't echo commands to be executed (equivalent to .SILENT). This option will stop armmake from outputting messages but it does not stop the commands it issues from outputting messages.
`-t`	Generate commands to make target(s) up-to-date by setting source time stamps consistently (only guaranteed to succeed if all sources exist).

Notes

The list of targets and macro definitions is space-separated, and optional. List elements are:

- deemed a target to be built by armmake if they contain no '=' character. The target is added to the list of objects to be built by armmake. If no targets are specified, the first target in the makefile is built.
- deemed to be a macro definition if they contain an '=' character. The string to the left of the '=' is interpreted as a macro name, which is initialised to be the string to the right of the '=' character. Note that if any spaces are to be specified in the value of the macro, the whole list element should be enclosed in double quotes.
  For example:
  ```
  armmake myprog "STRING=Three Blind Mice"
  ```

13.3 Makefile Format for armmake

An armmake makefile consists of a sequence of logical lines. A logical line may be continued over several physical lines provided each but the last line ends with a `\`.

Comments are ignored by armmake. A comment is introduced by a hash character (#) and runs to the end of the logical line. A literal ’#’ character can be produced by escaping it with a backslash thus: `\#`.

Other than comments, there are four kinds of non-empty logical lines in a makefile:

- dependency lines
- command lines
- macro definition lines
- rule and other special lines (see *13.5.3 Rule patterns, .SUFFIXES, $@, $*, $< and $?* on page 13-8).

13.3.1 Dependency lines

Dependency lines have the form:

```
space-separated-target-list : space-separated-prerequisite-list.
```

For example:

```
program.exe : module1.obj module2.obj module3.obj library.lib
```

A dependency line cannot begin with white space. Spaces before the ':' are optional, but some white space must follow the ':', to distinguish ':' separating targets and prerequisites from ’:’ as part of a filename.

A line with multiple targets is shorthand for several lines, each with one target and the same righthand side (and the same associated commands, if any).

Multiple dependency lines referring to the same target accumulate, though only one such line may have commands associated with it (armmake would not know in what order to execute the commands otherwise). For example, the above dependency line could be rewritten as several dependency lines:

```
program.exe : module1.obj
program.exe : module2.obj
program.exe : module3.obj
program.exe : library.lib
```

13.3.2 Command lines

Command lines immediately follow a dependency line and begin with white space.

For maximum compatibility with UNIX makefiles, ensure that the first character of every command line is a tab.

A semi-colon may be used instead of a new line to introduce commands on a dependency line. This is most often useful when there are no prerequisites and only a single command associated with a target. For example:

```
clean:; del *.obj
```

Note that in this case no white space is needed between the colon and the semicolon.

13.3.3 Macro definition lines

Macro definition lines have the form:

```
macro-name = some-text-to-the-end-of-the-logical-line
```

For example:

```
CC = armcc
CFLAGS = -li -apcs 3/32bit -fah -c
LINK = armlink
LIB = \release\lib\armlib.32l
```

The '=' can be surrounded with white space, or not, to taste. Thereafter, wherever `$(`*name*`)` or `${`*name*`}` is encountered, the whole of `$(`*name*`)` or `${`*name*`}` is replaced by the corresponding definition. A reference to an undefined macro simply vanishes. An example that uses the above macro definitions is:

```
program:  program.obj $(LIB)
      $(LINK) -o program $(LFLAGS) program.obj $(LIB)
```

Note that `$(LFLAGS)` expands to nothing, as it is undefined.

Macros can also be defined on armmake's command line as described in *13.2 Command-line Options* on page 13-3.

13.4 Command Execution

When attempting to build a target, armmake uses all the dependency lines specifying the current target as a target, and generates a list of all the corresponding prerequisites. Each of these prerequisites is checked, and if they are not up-to-date will be made up-to-date before any further processing is performed. Finally, the commands on the command lines for the current target are executed.

These commands are executed sequentially. If a command returns a non zero return code armmake will normally stop processing, taking this to mean that the build has failed.

Note that there is an MSDOS command length limit of 127 characters. If this is exceeded armmake will give a warning message. This limit may cause problems, particularly with armlink, which is often passed many object files to link together. However, armlink can read its object files from a file, thus avoiding the problem. (See *6.2 Using the Linker* on page 6-3 for more information.) In addition, batch files containing complicated or long commands can be executed from the command lines.

13.5 Advanced Features

13.5.1 File naming

To help you use UNIX makefiles with armmake under MSDOS, armmake accepts both UNIX and MSDOS filenames, internally converting them to the host system style. Thus all of the following are acceptable:

UNIX-like:	`/tools/prog/test.c`	`../include/defs.h`
MSDOS-like:	`\tool\prog\test.c`	`..\include\defs.h`

In addition to filename interpretation on the dependency lines, armmake attempts to convert files on the command lines too. This is not straightforward as command lines may contain anything. armmake only changes a suspected filename if the filename has already been found on a dependency line in the file.

13.5.2 VPATH

Usually armmake looks for files relative to the current directory or in places implicit in the filename. The previous example contains the line:

```
program:  program.obj $(LIB)
```

which refers to `.\program.obj`

Sometimes, particularly when dealing with multiple versions of large systems, it is convenient to have a complete set of object files locally, a few sources locally, but most sources in a central place shared between versions.

If the macro VPATH is defined, armmake will look in the list of places defined in it for any files it can't find in the places implied by their names.

For example, we might have sources in \shared and \oldvsn. In order to tell armmake to look in these places if the normal search fails, use the following VPATH setting:

```
VPATH = \shared \oldvsn
```

Unlike UNIX VPATH's, the path elements are separated by spaces rather than colons.

Note: VPATH is not applied to any target names.

13.5.3 Rule patterns, .SUFFIXES, $@, $*, $< and $?

All the examples given so far have used explicit rules for making targets. In fact, armmake can infer rules for if you supply it with appropriate rule patterns. These are specified from a pseudo dependency called `.SUFFIXES` using special target names consisting of the concatenation of two suffixes.

An example is given below:

```
.SUFFIXES:      .obj .c
program:        program.obj $(LIB)
.c.obj:;        $(CC) $(CFLAGS) -o $@ $*.c
```

The rule pattern `.c.obj` describes how to make `.obj` files from `.c` files. If, as in the above fragment, there is no explicit entry describing how to make a particular `.obj` file (`program.obj` in the above example) armmake will apply the first rule it has for making `.obj` files. Here, order is determined by order in the `.SUFFIXES` pseudo-dependency.

Suppose `.SUFFIXES` were defined as `.obj .c .s` and that there were two rules, `.c.obj:...` and `.s.obj: ...` armmake would choose the `.c.obj` rule because `.c` precedes `.s` in the `.SUFFIXES` dependency. In applying the `.c.obj` rule, armmake infers a dependence on the corresponding `.c` file (here `program.c`). So, in effect, it infers:

```
program.obj:  program.c
                 $(CC) $(CFLAGS) -o program.obj program.c
```

Note that $@ is replaced by the name of the target and $* by the name of the target with the extension deleted from it. In a similar fashion, $< refers to the list of inferred prerequisites. So the above example could be rewritten using the rule:

```
.c.obj:;        $(CC) $(CFLAGS) -o $@ $<
```

However, if a VPATH were being used, this second form is obligatory. Consider, for example, the fragment:

```
VPATH = \shared \oldvsn
program:      ... module.obj ...
.c.obj:;      $(CC) $(CFLAGS) -o $@ $<
```

There is no explicit rule for making `module.obj`, so armmake will apply the rule pattern `.c.obj`. This might expand to:

```
module.obj:   \shared\module.c
              $(CC) $(CFLAGS) -o module.obj \shared\module.c
```

Clearly $* could not have been used in this case.

Finally, $? can be used in any command to stand for the list of prerequisites with respect to which the target is out of date (which may be only some of the prerequisites).

13.5.4 Use of ::

If you use ‘::’, rather than ‘:’, to separate targets from prerequisites, the righthand sides of dependencies which refer to the same targets are not merged. Furthermore, each such dependency can have separate commands associated with it. Consider, for example:

```
t1.obj::   t1.c t1.h
           armcc -g -c t1.c     # executed if t1.obj is out of
                                # date wrt t1.c or t1.h
t1.obj::   t1.c t2.h
           armcc -c t1.c        # executed if t1.obj is out of
                                # date wrt t1.c or t2.h
```

13.6 Miscellaneous Features

The special pseudo-target `.SILENT` tells armmake not to echo commands to be executed to your screen. Its effect is as if you used `armmake -s`.

The special pseudo-target `.IGNORE` tells armmake to ignore the return code from the commands it executes. Its effect is as if you used `armmake -i`.

A command line, the first non-white-space character of which is @, is locally silent; the command is not echoed.

A command line, the first non-white-space character of which is ‘-’ has its return code ignored when it is executed. This is extremely useful in makefiles which use commands that do not set the return code conventionally. Note particularly that for built-in commands under DOS, the return codes are meaningless, and as such should not be relied upon to stop makefile execution when they fail.

The special macro MFLAGS is given the value of the command line arguments passed to armmake. This is most useful when a makefile itself contains armmake commands (for example, when a system consists of a collection of subsystems, each described by its own makefile). MFLAGS allows the same command-line arguments to be passed to every invocation of armmake—even the recursive ones.

14 ARMulator

This chapter describes the ARM instruction set simulator.

ARMulator

14.1 About the ARMulator

The ARMulator is a family of programs which emulate the instruction sets of various ARM processors and their supporting architectures. The programs are written in C, and interface directly to user-supplied models written in C or C++. An ARMulator suitably equipped to support timing accuracy can be integrated into any hardware modelling system that accepts models written in C.

The ARMulator:

- provides an environment for the development of ARM-targeted software on a range of non ARM-based host systems
- allows accurate benchmarking of ARM-targeted software (though its performance will be somewhat slow compared to real hardware)
- supports the simulation of prototype ARM based systems, ahead of the availability of real hardware, so that software and hardware development can proceed in parallel
- provides a generic ARM processor core model for incorporation into hardware simulation environments (via the foreign language interfaces to those environments)

ARMulator can be transparently connected to the ARM symbolic debugger to provide a hardware-independent ARM software development environment. Communication takes place via the Remote Debug Interface (RDI): see ➲*Chapter 22, Remote Debugging* for further information. ARMulator also supports a full ANSI C library to allow complete C programs to run on the emulated system.

14.2 Modelling an ARM-Based System

Three levels of modelling accuracy can be supported:

- Instruction-accurate
- Cycle-accurate
- Timing-accurate

The rest of this chapter describes instruction-accurate emulation. This models the instruction set without regard to the precise timing characteristics of the processor. As a result, it is considerably faster than the other variants, and is best suited to software development and benchmarking.

14.3 ARMulator Release Components

The `armul` directory of the ARM Software Toolkit release contains the following files:

`Makefile`	Used to build the symbolic debugger
`readme`	This directs you to read this manual
`armfast.c`	ARMulator ‘fast’ memory model (small contiguous memory)
`armvirt.c`	ARMulator ‘virtual’ memory model (chunked memory)—the default
`armproto.c`	ARMulator ‘prototype’ memory model
`hostos.c` `hostos.h`	Support functions and headers for C-library on PIE Card
`armos.c` `armos.h`	Support functions and headers for C-library on ARMulator
`serdrive.c` `serdrive.h`	Source and headers for serial line driver for PIE Card
`armcopro.c`	ARMulator coprocessor model

`aif.h`
`armdefs.h`
`armfpe.h`
`dbg_conf.h`
`dbg_hif.h`
`pirdi.h`
	Assorted files needed to build the symbolic debugger

`armsd.a`	symbolic debugger library
`armsd`	armsd executable—placed in binaries directory (not `armul`)

The ARMulator is not a stand-alone product. To use it, you must link the parts together with a debugger to produce an executable file. The debugger enables you to load and execute ARM Image Format images.

The ARMulator can be customised by modifying the supplied C source files.

14.4 Building an ARMulator Variant

To build a debugger with options different to those provided in the release, modify the source files appropriately. Then select `armul` as the current working directory and enter one of the following commands:

`make`	Builds the symbolic debugger
`make armsd`	Builds the command-line debugger
`make MODEL=armfast`	Builds both, using the 'fast' memory model

Any customised source files (eg. `armfast.c`, `armos.c`, `serdrive.c`) will be automatically recompiled by the Makefile and included in the generated debugger executables.

14.5 Memory Interfacing

For instruction and program based emulation, a memory interface must be constructed to allow ARMulator to load instructions and access data. The interface can be constructed in two ways.

- Rapid prototype interface. This is designed to allow rapid prototyping of systems, and requires the construction of only three routines:
  ```
  ARMul_MemoryInit
  ARMul_MemoryExit
  ARMul_MemAccess
  ```
  See *14.8.2 Rapid prototype memory model* on page 14-10.
- High speed memory interface. This form requires the construction of 25 separate routines for handling instruction and data accesses, word and byte accesses, and so forth.

  See *14.8.1 High-speed memory interface functions* on page 14-6.

14.5.1 Coprocessor interfacing

ARMulator defines two ways of modelling a coprocessor:

- by accessing the modelled ARM's state directly through the `ARMul_State` structure
- by building a software interface that uses ARMulator to handle the difficult handshaking and timing constraints of the simulation environment

14.5.2 Event scheduling

There are two routines to assist with the handling of asynchronous events:

`ARMul_Time`	returns the number of clock ticks executed since system reset, modulo 2^{32}.
`ARMul_ScheduleEvent`	allows a function to be called a number of clock ticks into the future, thus enabling code such as multi-cycle floating point instructions to present their results at the appropriate time.

14.5.3 Adding an operating system

The function `ARMul_HandleSWI` supports rapid prototyping of operating systems. The SWI numbers of all executed SWI (software interrupt) instructions are passed to this function, allowing support code to perform system-specific operations. The function may refuse to handle a SWI by returning FALSE, in which case a SWI exception will occur. If the function returns TRUE, execution will continue normally after the SWI instruction has completed.

14.6 ANSI C Library

A full ANSI C Library is available to support the emulation of complete C programs with ARMulator. This library uses the ARMulator Operating System to perform filing operations on the emulation host, and an emulator for full floating-point support.

14.7 The ARMulator Environment

An ARMulator consists of four parts:

1. the ARM processor model, together with the Remote Debug Interface and utility functions to support system modelling
2. a memory interface which transfers data between the ARM model and the memory model or memory management unit model
3. a co-processor interface to optional ARM co-processor models
4. an operating system interface to provide an execution environment.

Part 1 incorporates the Remote Debug Interface (RDI), which connects the ARMulator to the symbolic debugger, in order to provide a hardware-independent, low-level software development environment: see *22.1 ARM Remote Debug Interface* on page 22-2.

Sample models of memory, a co-processor, and a simple operating system are supplied in source form with the ARMulator.

The file `armdefs.h` defines functions, data structures and useful macros to support modelling.

Floating point support

On start-up, the host debugger loads a software floating-point emulator into the ARMulator. Under armsd, this can be overridden using the `–nofpe` option. Programs using the software floating point library do no need this emulator to be loaded.

14.8 Memory Models

For instruction-based and program-based emulation, a memory model must be constructed to allow the ARMulator to load instructions and access data. A model can be built at two levels, the first designed for rapid prototyping and the second for maximum emulation performance.

An example of each type of model is supplied in source form with the ARMulator.

The `ARMul_State` structure contains four pointers (type `char*` in C) that the memory interface can use to refer to the memory model:

`MemDataPtr`	is a pointer to the memory system's private data structures, for example to a Translation Lookaside Buffer.
`MemInPtr`	is used to refer to memory attached to ARM's DataInBus
`MemOutPtr`	used to refer to memory attached to ARM's DataOutBus
`MemSparePtr`	is free for any extra data that may require referencing

If the multiple instantiation property of the ARMulator is to be preserved, it is essential that memory models declare no static data of their own, but chain dynamically allocated (`malloc`'d) state-containing structures off `MemDataPtr`.

A fifth field in `ARMul_State` structure, `MemSize`, is used by the C runtime system to find the top of RAM memory. If this value is set, the C user stack will descend from this address, otherwise the runtime system will use a default value.

14.8.1 High-speed memory interface functions

To maximise the performance of a memory model (thus maximising the number of instructions emulated per second), the following high speed memory interface should be used. This interface can distinguish between different sorts of memory access using different access functions.

Twenty five functions have to be implemented (in C). These are described on the following pages, grouped according to function.

Initialisation/finalisation

```
unsigned ARMul_MemoryInit(ARMul_State *state,
                          unsigned long memorysize)
```

This function is called once, before any other routines are called, to allow the memory manager to initialise itself. If initialisation fails, the routine should return FALSE, otherwise it should return TRUE.

The `memorysize` argument specifies the minimum amount of memory required by the ARMulator. If this cannot be supplied, initialisation should fail. For systems implementing a fixed size memory, `memorysize` should be used to set the value of `MemSize` in `ARMul_State`. The memory model can use this opportunity to attach to other parts of the ARMulator—an MMU model would probably want to attach to coprocessor 15 and the ModeChange upcall, for example.

```
unsigned ARMul_MemoryExit(ARMul_State *state,
                          unsigned long memorysize)
```

This function is called once to allow the memory manager to perform finalisation code (for tasks such as deallocating memory). If finalisation fails, the function should return FALSE, otherwise it should return TRUE.

Load instruction

```
ARMword ARMul_LoadInstrS(ARMul_State *state, ARMword address)
```

This function should return the instruction addressed by `address`. The address was produced by the address incrementer, and sequentially follows that of the preceding memory access.

```
ARMword ARMul_LoadInstrN(ARMul_State *state, ARMword address)
```

This function should return the instruction addressed by `address`. The address does not sequentially follow that of the preceding memory access.

```
ARMword ARMul_LoadInstr16S(ARMul_State *state, ARMword address)
```

This function loads 16-bit (Thumb) instructions, and should return the value in the lower 16 bits of the result. The address was produced by the address incrementer, and sequentially follows that of the preceding memory access.

```
ARMword ARMul_LoadInstr16N(ARMul_State *state, ARMword address)
```

This function loads 16-bit (Thumb) instructions, and should return the value in the lower 16 bits of the result. The address does not sequentially follow that of the preceding memory access.

Load data

```
ARMword ARMul_LoadWordS(ARMul_State *state, ARMword address)
```

This function should return the word addressed by `address`. The address was produced by the address incrementer and sequentially follows that of the preceding memory access.

```
ARMword ARMul_LoadWordN(ARMul_State *state, ARMword address)
```

This function should return the word addressed by `address`. The address does not sequentially follow that of the preceding memory access.

```
ARMword ARMul_LoadByte(ARMul_State *state, ARMword address)
```

This function should return the byte addressed by `address`. The byte should be returned in the least significant eight bits of the returned value.

```
ARMword ARMul_LoadHalfWord(ARMul_State *state, ARMword address)
```

This function loads a halfword, and returns the value in the least significant sixteen bits of the returned value.

Store data

```
void ARMul_StoreWordS(ARMul_State *state, ARMword address,
                      ARMword data)
```

This function stores the word `data` to memory at address `address`. The address was produced by the address incrementer and sequentially follows that of the preceding word stored.

```
void ARMul_StoreWordN(ARMul_State *state, ARMword address,
                      ARMword data)
```

This function stores the word `data` to memory at address `address`. The address does not sequentially follow that of the preceding word stored.

```
void ARMul_StoreByte(ARMul_State *state, ARMword address, ARMword data)
```

This function stores the byte in the least significant eight bits of `data` to memory, at address `address`.

```
void ARMul_StoreHalfWord(ARMul_State *state, ARMword address,
                         ARMword data)
```

This function stores the byte in the least significant sixteen bits of `data` to memory, at address `address`.

Swap

```
ARMword ARMul_SwapByte(ARMul_State *state, ARMword address
                        ARMword data)
```

This function should load the byte addressed by `address`, store the byte from the least significant eight bits of `data` at `address`, and return the loaded value.

```
ARMword ARMul_SwapWord(ARMul_State *state, ARMword address,
                        ARMword data)
```

This function should load the word addressed by `address`, store the word `data` at address, and return the loaded value.

Read

```
ARMword ARMul_ReadByte(ARMul_State *state, ARMword address)
ARMword ARMul_ReadWord(ARMul_State *state, ARMword address)
ARMword ARMul_ReadHalfWord(ARMul_State *state, ARMword address)
```

These three functions must be *pure* and must transfer data from the memory model without updating any other internal state. Both are used by implementations of the Remote Debug Interface to inspect the modelled memory (see ➲*22.1 ARM Remote Debug Interface* on page 22-2).

As a general rule, these calls should reflect the same memory mapping as that seen by the processor in its current state, but should not, for example, cause cache line-fetches.

Write

```
void ARMul_WriteByte(ARMul_State *state, ARMword address, ARMword data)
void ARMul_WriteWord(ARMul_State *state, ARMword address, ARMword data)
void ARMul_WriteHalfWord(ARMul_State *state, ARMword address,
                          ARMword data)
```

These three functions must be *pure* and must transfer data to the memory model without updating any other internal state. Both are used by implementations of the Remote Debug Interface to alter the modelled memory (see ➲*22.1 ARM Remote Debug Interface* on page 22-2).

As a general rule, these calls should reflect the same memory mapping as that seen by the processor in its current state, but should not enforce protection attributes, nor account for the accesses.

Execute cycles

```
void ARMul_Ccycles(ARMul_State *state, unsigned number,
                   ARMword address)
```

This function is used to inform the memory system that the processor is about to execute `number` co-processor cycles.

```
void ARMul_Icycles(ARMul_State *state, unsigned number,
                   ARMword address)
```

This function is used to inform the memory system that the processor is about to execute `number` internal (I) cycles. The address of the next memory access will be `address`.

Attach mode change handler

```
void ARMul_ModeChangeUpcall(ARMul_State *state, ARMword old,
                            ARMword new)
```

The memory system is at liberty to attach a handler to the ModeChange upcall. This is a function pointer in `ARMul_State`, and is called every time the processor mode is changed. This allows access permissions to be switched. The handler should take a copy of the old pointer value, and pass the call on to that function when it has finished (the default value of the pointer is a do-nothing function). The function is called as shown above, immediately after the register banks have been switched (if appropriate).

14.8.2 Rapid prototype memory model

Providing the ARMulator with a prototype memory model involves writing just three C functions. (Veneer implementations of the functions listed in the previous section are supplied with ARMulator in source code form.)

The three functions are as follows.

```
unsigned ARMul_MemoryInit(ARMul_State *state, unsigned long memorysize)
```

This function is called once, before any other routine, to allow the memory manager to initialise itself. If initialisation fails, the routine should return FALSE, otherwise it should return TRUE.

The `memorysize` argument specifies the minimum amount of memory required by the ARMulator. If the size cannot be supplied, initialisation should fail. For systems implementing a fixed size memory, `memorysize` should be used to set the value of `MemSize` in `ARMul_State`.

```
unsigned ARMul_MemoryExit(ARMul_State *state, unsigned long memorysize)
```

This function is called once to allow the memory manager to perform finalisation code (for tasks such as deallocating memory). If finalisation fails, the function should return FALSE, otherwise it should return TRUE.

```
ARMword ARMul_MemAccess(
       ARMul_State *state,
       ARMword address,
       ARMword dataOut,
       ARMword mas1,
       ARMword mas0,
       ARMword rw,
       ARMword seq,
ARMword mreq,
ARMword opc,
ARMword lock,
ARMword trans,
ARMword account)
```

This routine must perform a memory access by modelling the ARM memory interface pins.

The parameters following the `state` parameter have the following meanings:

`address`	encodes ARM's address bus and always has a valid value.
`dataOut`	encodes ARM's **dataOut** bus and only has a valid value during memory write requests.
`mas1 mas0`	encode ARM's **mas[1:0]** bus, which encodes the width of the data transfer requested. Possible values are:

mas1	mas0	
0	0	byte
0	1	halfword
1	0	word
1	1	reserved

Its value is only valid when the value of `mreq` (see below) is LOW, ie, during data transfer cycles.

`rw`	represents ARM's Not Read/Write pin, and has a value of LOW for read requests and HIGH for write requests. Its value is only valid when `mreq` (see below) is LOW (ie. during data transfers). Byte read requests should return the entire word from the memory. The ARMulator will extract the required byte, just as ARM itself would. Byte Write operations receive the byte to be written replicated four times in `dataOut`; the memory model must extract the correct byte.
`seq`	represents ARM's Sequential pin, and has a value of HIGH when the address specified is sequential with the previous access. Its value is always valid.
`mreq`	represents ARM's Not Memory Request Pin, and has a value of LOW when the access should actually load or store data. Its value is always valid.
`opc`	represents ARM's Not OpCode Pin, and has a value of LOW if the access is an instruction fetch. Its value is only valid when the value of `mreq` (see above) is LOW (ie. during data transfers).
`lock`	represents ARM's Lock Pin, and has a value of HIGH if the memory system should deny other devices access to the memory. The signal is only valid when `mreq` (see above) is LOW (ie. during data transfers).
`trans`	represents ARM's Not Trans Pin, and has a value of LOW if the processor is in a User mode. Its value is always valid.

`account`	is used to inform the memory system that it should account for the number of cycles used in this memory access by updating the fields `NumScycles`, `NumNcycles`, `NumIcycles` and `NumCcycles` in the `ARMul_State` structure pointed to by `state`.

When `rw` is LOW, `ARMul_MemAccess` should return the contents of the memory location addressed by `address`, otherwise it should return 0. Aborts in the memory system are modelled using two macros defined in `armdefs.h`.

If an Abort occurs during an instruction prefetch (`opc` equals LOW), the macro `ARMul_PREFETCHABORT` should be called, passing the address of the abort (from the address argument to `ARMul_MemAccess`) as a parameter.

If a Data Abort occurs, the macro `ARMul_DATAABORT` should be called, again using address as its parameter. In both cases the value ARMul_ABORTWORD should be returned.

14.9 Co-processor Modelling

The ARMulator models a co-processor via a software interface that uses the ARMulator to handle most of the difficult handshaking and timing constraints. Each co-processor modelling function must be installed in the `ARMul_CoProInit` function defined in `armcopro.c` using the function `ARMul_CoProAttach`, and de-installed using the function `ARMul_CoProDetach`.

14.9.1 Installing a co-processor model

Each co-processor modelling function is installed using the function `ARMul_CoProAttach`:

```
void ARMul_CoProAttach(ARMul_State *state, ARMword number,
                          ARMul_CPInits *init, ARMul_CPExits *exit,
                          ARMul_LDCs *ldc, ARMul_STCs *stc,
                          ARMul_MRCs *mrc, ARMul_MCRs *mcr,
                          ARMul_CDPs *cdp,
                          ARMul_CPReads *read, ARMul_CPWrites *write,
                          ARMword *regwords)
```

This installs up to nine functions for co-processor number `number`. The remaining arguments are pointers to functions (see ➲*14.9.2 Co-processor modelling functions*, below). If an argument is NULL, the relevant handler is not changed.

The `regwords` argument describes to ARMulator the shape of your co-processor's registers. It consists of a single word giving the number of registers followed by a vector of the minimum number of words required to contain the register. For an example, see the definition of the minimal MMU system in `armos.c`.

The state structure contains the field:

```
char *CPData[16]
```

which can be used by a co-processor to point to private data. Each co-processor should use the element of the array corresponding to its co-processor number.

14.9.2 Co-processor modelling functions

The following functions may be passed to `ARMul_CoProAttach`.

Initialisation/finalisation functions

```
unsigned ARMul_CPInits(ARMul_State *state)
```

This function is called once, before any other routine, to allow the co-processor to initialise. If initialisation is not possible, the routine should return FALSE, otherwise it should return TRUE.

```
unsigned ARMul_CPExits(ARMul_State *state)
```

This function is called last, allowing the co-processor to finalise. If finalisation is not possible, the routine should return FALSE, otherwise it should return TRUE.

Co-processor read/write functions

```
unsigned ARMul_CDPs(ARMul_State *state, unsigned type, ARMword instr)
```

This function is called whenever the ARMulator encounters a CDP instruction destined for this co-processor. Unless it returns the value BUSY, it will only be called once with `type` set to FIRST: see *Read/write function parameters* on page 14-14.

```
unsigned ARMul_MRCs(ARMul_State *state,
                      unsigned type, ARMword instr, ARMword *value)
```

This function is called whenever the ARMulator encounters an MRC instruction destined for this co-processor.

The `value` argument is a pointer to (the model of) an ARM register where the result of the transfer should be stored (if it was successfully emulated). This location should be considered write-only. Unless the function returns the value BUSY, it will only be called once with `type` set to `FIRST`: see *Read/write function parameters* on page 14-14 .

```
unsigned ARMul_MCRs(ARMul_State *state,
                      unsigned type, ARMword instr, ARMword value)
```

This function is called whenever the ARMulator encounters an MCR instruction destined for this co-processor. The `value` argument is the value of the ARM register to transfer (if it was successfully emulated). Unless the function returns BUSY, it will only be called once with `type` set to `FIRST`: see *Read/write function parameters* on page 14-14.

```
unsigned ARMul_LDCs(ARMul_State *state,
                      unsigned type, ARMword instr, ARMword value)
```

This function is called whenever the ARMulator encounters an LDC instruction destined for this co-processor. The `value` argument is the data loaded from memory. This function will be called first with `type` set to `FIRST`. Assuming the function does not return BUSY, the function will then be called with `type` set to TRANSFER, and finally with `type` set to DATA, at which point the `value` argument will be valid. The function may request further data by returning the value `INC`, in which case subsequent calls will have `type` set to `DATA` and the `value` argument will be valid: see *Read/write function parameters* on page 14-14.

```
unsigned ARMul_STCs(ARMul_State *state,
                    unsigned type, ARMword instr, ARMword *value)
```

This function is called whenever the ARMulator encounters an STC instruction destined for this co-processor. The `value` argument should be set to the value to be stored to memory. This location should be considered write-only.

`ARMul_STCs` will be called first with `type` set to `FIRST`. Assuming it does not return `BUSY`, the function will then be called with `type` set to `DATA`, at which point it should store the data to be written to memory in `value`.

The function may request further data to be transferred by returning the value `INC`. Subsequent calls will have `type` set to `DATA` and `value` should be set to the value to be written to memory: see *Read/write function parameters* on page 14-14.

Read/write function parameters

The functions listed in the previous section accept a parameter called `type`. This can have one of five values:

`FIRST`	indicates that this is the first time the co-processor has been called with this instruction.
`TRANSFER`	indicates that this is the load cycle of an LDC instruction (the data is being loaded from memory).
`DATA`	indicates that the co-processor is being called with valid data (LDC and MCR), or is expected to return valid data (STC and MRC).
`INTERRUPT`	is used to warn the co-processor that the ARMulator is about to service an interrupt, so the co-processor should discard the current instruction. Usually, the instruction will be retried later, in which case the `type` parameter will be reset to `FIRST`.
`BUSY`	is used as the reply to a previous call that returned BUSY (see below)

The `instr` parameter is the co-processor instruction itself.

The functions must return one of four values:

`BUSY`	indicating that the co-processor is busy and the ARMulator should busy-wait, recalling the routine repeatedly
`CANT`	indicating that the co-processor cannot execute this particular instruction
`INC`	indicating that the ARMulator should produce the next address for an LDC or STC instruction, and call the co-processor function again
`DONE`	indicating successful completion of the instruction.

Debug functions

Finally, two functions allow a debugger to read and write co-processor registers via the Remote Debug Interface (see ➲*22.1 ARM Remote Debug Interface* on page 22-2):

```
unsigned ARMul_CPReads(ARMul_State *state,
                      unsigned reg, ARMword *value)
```

This reads the co-processor register numbered `reg`, and transfers its value to the location addressed by `value`.

```
unsigned ARMul_CPWrites(ARMul_State *state,
                      unsigned reg, ARMword *value)
```

This sets the value of the co-processor register numbered `reg` to the value addressed by value.

14.9.3 De-installing a co-processor

A co-processor is de-installed using the following function.

```
extern void ARMul_CoProDetach(ARMul_State *state, ARMword number)
```

This sets the default handlers for co-processor number `number`. These consist of no initialisation, finalisation, read or write functions, and dummy operation functions which cause ARM undefined instructions.

14.10 Modelling an Operating System or Low Level Monitor

Rapid prototyping of low-level operating system code is supported via the following four functions in the file `armos.c`. In all cases, the `ARMul_State` structure pointed to by `state` contains a pointer called `OSptr`, which can be used to point to a structure holding the operating system model's state.

```
unsigned ARMul_OSInit(ARMul_State *state)
```

`ARMul_OSInit` informs the OS model that it should initialise itself. The function may return FALSE, indicating that initialisation was impossible. The memory system is guaranteed to be operating at this time, and thus the memory interface routines described in ➲*14.8 Memory Models* on page 14-6 may be used to install default trap handlers, etc.

```
unsigned ARMul_OSHandleSWI(ARMul_State *state, unsigned SWInum)
```

`ARMul_OSHandleSWI` is passed the SWI number (in `SWInum`) of each SWI instruction executed by the ARMulator, as it is executed, allowing support code to simulate operating system operations. You can extend this code to model as much of your operating system as you choose. This OS model can, of course, communicate with the emulated application, by reading and writing the emulated ARM state using the routines described in ➲*14.11 Accessing ARMulator's State* on page 14-17.

The function may refuse to handle a SWI by returning FALSE, in which case the SWI exception vector is taken by the ARMulator. If the function returns TRUE, execution continues normally once the SWI has completed.

```
ARMword ARMul_OSLastErrorP(ARMul_State *state)
```

This function should return the last software error reported to the debuggee via a default Error Vector handler. If no handler is installed, the routine should return zero.

```
unsigned ARMul_OSException(ARMul_State *state, ARMword vector,
                           ARMword pc)
```

`ARMul_OSException` 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. The argument `vector` contains the address of the vector about to be executed: 0 for Reset, 4 for Undefined Instruction, 0x1c for Fast Interrupt (FIQ) and so on. The argument pc contains the value of 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 execution will continue from the instruction following the aborted instruction. A return value of TRUE will cause the exception to occur normally.

14.11 Accessing ARMulator's State

The ARMulator provides several functions that allow full access to its internal state. These may only be called at the beginning of a new instruction—ie. from:

- any initialisation or finalisation function
- `ARMul_OSHandleSWI`, `ARMul_LoadInstrS` and `ARMul_LoadInstrN` (or `ARMul_OSHandleSWI`, `ARMul_LoadInstr16S` and `ARMul_LoadInstr16N` for the 16-bit versions)
- a routine called by the ARMulator's event handler: see *14.14 Event Handling* on page 14-18

At any other time the ARMulator's state is undefined.

```
ARMword ARMul_GetReg(ARMul_State *state, unsigned mode, unsigned reg)
void ARMul_SetReg(ARMul_State *state, unsigned mode, unsigned reg)
```

These functions allow the value of a register from a given mode to be read and written. Register 15 should not be accessed with these functions.

The mode numbers are defined in `armdefs.h` as follows:

```
USER26MODE                USER32MODE
FIQ26MODE                 FIQ32MODE
IRQ26MODE                 IRQ32MODE
SVC26MODE                 SVC32MODE
                          ABORT32MODE
                          UNDEF32MODE
```

```
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/preserve 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 and written. The format of the CPSR is the same as that of ARM6, regardless of whether the processor is in 26 bit mode or not.

```
ARMword ARMul_GetSPSR(ARMul_State *state, ARMword mode)
void ARMul_SetSPSR(ARMul_State *state, ARMword mode, ARMword value)
```

These functions allow the SPSR of a specified mode to be read and written. The format of an SPSR is the same as that of ARM6, regardless of whether the processor is in 26 bit mode or not.

14.12 ARMulator Signals

Some of ARM's I/O signals can be read and written at any time by accessing the fields of the `ARMul_State` structure. The following fields are guaranteed to contain valid values at all times:

```
ARMword instr
ARMword pc
unsigned NresetSig
unsigned NfiqSig
unsigned NirqSig
unsigned abortSig
unsigned NtransSig
unsigned bigendSig
unsigned prog32Sig
unsigned data32Sig
unsigned lateabtSig
```

The `instr` field contains the address of the instruction that is currently being executed, the `pc` field contains the address of that instruction. The meaning of the other fields are described in the ARM datasheets, and should be set to the value HIGH or LOW. The signals `Nreset`, `Nfiq` and `Nirq` are only checked if `Exception` in the state structure is set to TRUE.

14.13 Processor Selection

The following function can be used to change the processor being emulated.The full range of possible processors is listed in `armdefs.h`.

```
void ARMul_SelectProcessor(ARMul_State *state, int processor)
```

This function will, as a side-effect, alter the values of various I/O signals.

Note: There is currently no way to specify processor requirements over the RDI. To specify a processor, use `ARMul_SelectProcessor` from within your `ARMulMemoryInit` implementation.

14.14 Event Handling

The ARMulator has two routines to assist with scheduling asynchronous events.

```
unsigned long ARMul_Time(ARMul_State *state)
```

This function returns the number of clock ticks executed since system reset, modulo 2^{32}.

```
void ARMul_ScheduleEvent(ARMul_State *state,
       unsigned howlong, unsigned (*func)(ARMul_State *state))
```

This function allows a function (passed in the argument `func`) to be called `howlong` clock ticks into the future, therefore allowing code like multicycle FPU instructions to produce results some time into the future. The function is called with a pointer to the emulation state pertaining at the time of the call. The value of `howlong` must not exceed 1000.

14.15 The ARM Debug Monitor

The files `armos.h`, `armfpe.h` and `armos.c` define a low level interface between the ARMulator and the host's operating system, used for debugging applications running under the ARMulator and for supporting the semi-hosted ANSI C library via the host's C library. The same interface is used between the ARM debug monitor (for platform independent ARM evaluation and development cards) and the ARM symbolic debugger. This gives a common low-level programming, C library and debugging environment across:

- the ARMulator under the symbolic debugger
- the ARM Platform Independent Evaluation (PIE) card for the ARM60 driven by the symbolic debugger

For further details, please refer to *Chapter 17, Demon*.

15 C Library

This chapter describes the ARM C library and how to port it to other targets.

C Library

15.1 An Introduction to the Run-Time Libraries

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

- the minimal standalone run time library
- the ANSI C library

Both libraries are supplied in source form and will need to be re-targeted to your ARM-based hardware. However, both have been designed to make re-targeting straightforward. Both libraries are also provided in binary form targeted at the ARMulator, ARM's user-extensible CPU emulator, so you can run and debug programs running on an emulated ARM immediately. For details see *Chapter 7, Symbolic Debugger*.

15.1.1 Minimal standalone run-time library

The minimal standalone run time library provides only those functions required by code compiled with the ARM C compiler:

- division and remainder functions
- stack-limit checking functions
- lowest-level memory management (stack + heap)
- program startup (calling `main()`)
- program termination (`_exit()`)

This library is a single module written in ARM assembly language. It is a few pages of code, a high proportion of which is comment. Only a few small functions need to be re-written for each new environment, and thus re-targeting typically takes an hour or two and is right first time.

Once the standalone library is operational, code compiled from C may be run, beginning with the supplied test programs.

15.1.2 ANSI C library

The full ANSI C library contains the following:

- target-independent modules written in ANSI C (eg. `printf()`);
- target-independent modules written in ARM assembly language (eg. `divide()`, `memcpy()`);
- target-dependent modules written in ANSI C (eg. default signal handlers, `clock()`);
- target-dependent modules written in ARM assembly language.

The target-independent portions of the library can be built immediately. The target-dependent parts require some effort to implement. An example implementation for a commercially available ARM-based personal computer is provided in the tools release, and this can be modified as required.

Re-targeting the ANSI C library will require some knowledge of ARM assembly language, and some understanding of the ARM processor and hardware being used. It will be necessary to refer to the relevant ARM datasheet, and also to *3.3 Assembly Language Overview* on page 3-6 and *Chapter 4, ARM Instruction Set*.

Because of its modular structure, not all of the library needs to be re-targeted at once: re-target only what will be used. Re-targeting the C library is described in section *15.2 Porting the ARM C Library*.

15.2 Porting the ARM C Library

The retargetable ARM C library conforms to the ANSI C library specification. Example code is included which targets the library at:

1. the common operating environment supported by the ARM emulator (ARMulator), the ARM Evaluation and Development boards
2. Acorn’s proprietary RISC OS operating system. (RISC OS is a cooperative multi-tasking system, but the targeting is at a level such that its multi-tasking nature does not obtrude.)

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

15.3 Source Organisation

The supplied source structure holds the following directories:

`stdh`	contains the ANSI header files (which should 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 semi-hosted 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 *semi-hosted* 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.
`riscos`	contains targeting code for Acorn's RISC OS operating system for its ARM-based computers, together with SunOS-hosted make information.
`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 mostly 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.

15.3.1 Constructing a makefile

The first stage in constructing a makefile for a targeting of the library uses the utility program makemake. This 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.

makemake takes as input the files `makedefs` from the host directory and `sources` and `options` from the target directory (see below for a description of their content). It produces as output a makefile called `Makefile` in the host directory (often, the host directory and the target directory will be the same).

The arguments to makemake are the name of the host directory and, if distinct, the name of the target directory.

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 ➲*15.5 Retargeting the Library* on page 15-7. 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, note). Use the file `sources` from the `semi` subdirectory as a template, modifying it as needed.

`hostsys.h` (in the target directory)

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 ➲*15.5 Retargeting the Library* on page 15-7 and ➲*15.6 Details of Target-Dependent Code* on page 15-10).

`config.h` (in the target directory)

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). Truly 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.

15.4 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
   ```
   (Since 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 DOS, use `util\makemake` ...)

3.
   ```
   cd hostdir
   make depend
   ```
   (this augments `Makefile`; as a side-effect it also makes the assembler-sourced objects.)

4.
   ```
   make
   ```
   (this makes `armlib.o` ... if everything succeeds).

15.5 Retargeting the Library

The following generic variants are available as ‘tick box’ options through 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), along with 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). In this case, if the target-dependent kernel follows the code of the riscos example, functions `__fp_initialise`, `__fp_finalise` and `__fp_address_in_module` must be provided (see ➲*15.6.3 Floating-point support* on page 15-12).
`=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 ➲*Chapter 16, Software Floating Point*.

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 obviously use more code. Complete unrolling of divide is possible, but should be done with care since the significant size increase might give decreased rather than increased performance on a cached ARM. Whichever variant is selected, fast unsigned and signed divide by 10 are included.

stack (see ➲*15.5.2 Address space model* on page 15-8).

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).

15.5.1 Basic choices

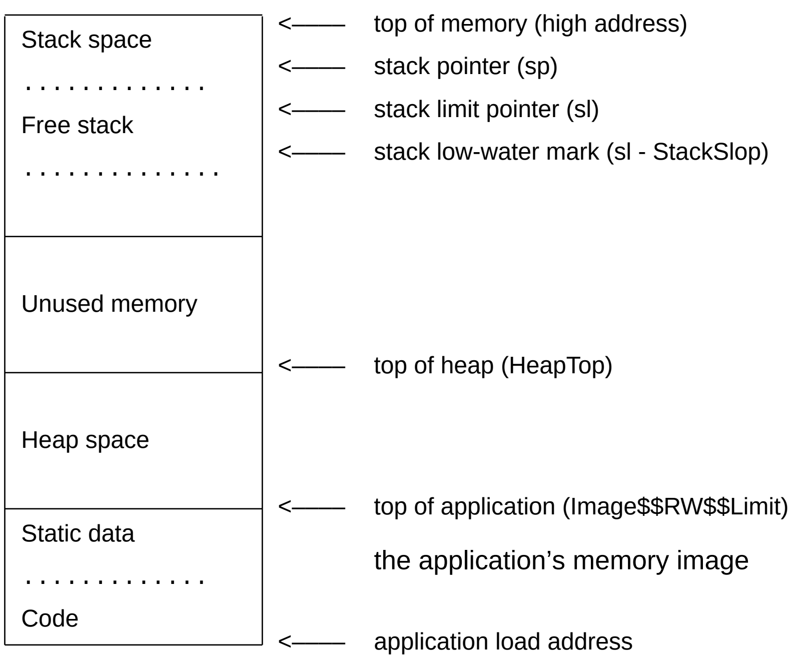
After the tick box choices have been made, basic choices then have to be made about the address-space model and the I/O model the library will follow.

15.5.2 Address space model

Two address space models are supported: contiguous stack and chunked stack.

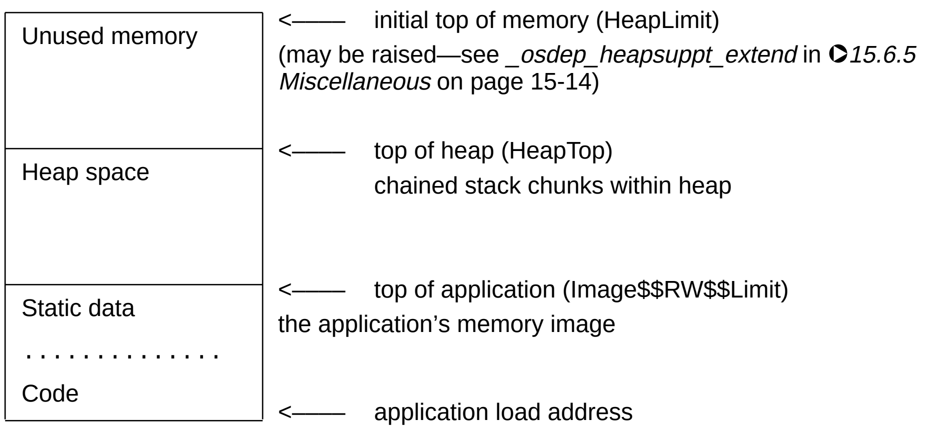
Contiguous stack

Choosing `stack` = `contiguous` gives:


Stack space
....
Free stack
....
Unused memory
Heap space
Static data
....
Code
<—— top of memory (high address)
<—— stack pointer (sp)
<—— stack limit pointer (sl)
<—— stack low-water mark (sl - StackSlop)
<—— top of heap (HeapTop)
<—— top of application (Image$$RW$$Limit)
the application's memory image
<—— application load address

Chunked stack

Choosing `stack = chunked` gives:


Unused memory
Heap space
Static data
Code
<—— initial top of memory (HeapLimit)
(may be raised—see _osdep_heapsuppt_extend in 15.6.5 Miscellaneous on page 15-14)
<—— top of heap (HeapTop)
chained stack chunks within heap
<—— top of application (Image$$RW$$Limit)
the application's memory image
<—— application load address

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

15.5.3 I/O model

The library, as supplied, only conveniently handles byte-stream files, (which is not to say that other file types cannot be handled in the target-dependent IO support level, but such support may well be complicated; block stream files, for example, are simple to support in the absence of user-supplied buffers).

15.6 Details of Target-Dependent Code

15.6.1 ANSI library functions

The following ANSI standard functions have an implementation completely dependent on the target operating system. No functions are used internally by the library (so if any are unimplemented, only clients which directly call the functions will fail).

```
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)
```

`clock_init()` (if you provide it) is called from the library's initialisation code, (`clock()` needs initialising 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 initialisation code if you provide an implementation of it.

15.6.2 I/O support

If any I/O function is to be used, `hostsys.h` must define the type `FILEHANDLE`, 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 to 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 disc if an OS write is followed by an OS read that requires a seek, (the flush happens before the seek). The RISC OS targeting needs this macro to be defined, thanks to an OS file-buffering bug; it is unlikely to be wanted otherwise.

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

`openmode` is a bitmap, whose bits mostly correspond directly to the ANSI mode specification: for details, see `hostsys.h` in ➲*15.3 Source Organisation* on page 15-4. (Target-dependent extensions are possible, in which case `freopen()` must be extended too). The function `_sys_open()` is needed by `fopen()` and `freopen()`, which in turn are required if any I/O function is to be used.

```
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)
```

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

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

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. This function must be defined if any output function or `sprintf` variant is to be used.

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

The mode argument is a bitmap describing the state of the FILE connected to `fh`, as for `_sys_write`. The return value is the number of characters *not* read (ie. `len - result` were read), or an error indication, or an EOF indicator. The target-independent code is capable of handling either early EOF (the last read from a file returns some characters plus an EOF indicator), or late EOF (the last read returns just EOF). The EOF indication involves the setting of 0x80000000 in the normal result. The function `_sys_read()` must be defined if any input function or `sscanf` variant is to be used.

```
int _sys_seek(FILEHANDLE fh, long pos)
```

This function positions the file pointer at offset `pos` from the beginning of the file. The result is >= 0 if OK, negative for an error. The function must be defined if any input or output function is to be used.

```
int _sys_ensure(FILEHANDLE fh)
```

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 OK, negative for an error. This function is only required if you define `HOSTOS_NEEDSENSURE` (see above).

```
long _sys_flen(FILEHANDLE fh)
```

The above 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 writes a character, notionally to the console. 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 (eg. in default SIGSTK handler). This function must be defined.

`int _sys_istty(FILE *)`

This function returns non-0 if the argument file is connected to a terminal. It is used to provide default unbuffered behaviour (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.

15.6.3 Floating-point support

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

If the variant `fp_type == module` is selected, and the target-dependent library kernel follows the pattern of the RISC OS example, these two functions must be supplied (though they need not do anything). The function `__fp_initialise()` returns 1 if floating-point instructions are available, otherwise 0.

`bool __fp_address_in_module(void *)`

If the variant `fp_type == module` is selected and the supplied abort handlers are used, the above function must be provided. 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).

15.6.4 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 are strongly dependent on details of 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()`

> The entry point to the application. Must perform low-level library initialisation, then call `_main`. (What initialisation needs to be done is target environment dependent: it may include heap, stack, and fp support, calling various `osdep_xxx_init()` functions if they exist). `__rt_lib_init` must be called to initialise the body of the library.

`void __rt_exit(int);`

> This function finalises the library (including calling `atexit()` handlers), then returns to the OS with its argument as a completion code. It must be provided.

`char *__rt_command_string(void);`

> This function returns the address of (maybe a copy of) the string used to invoke the program. It must be provided.

`void __rt_trap(__rt_error *, __rt_registers *);`

> `__rt_trap()` handles a fault (processor detected trap, enabled fp exception, or the like). 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). This function must be provided. The implementation in the ARMulator kernel will usually be adequate.

`unsigned __rt_alloc(unsigned minwords, void **block);`

> `__rt_alloc()` is the low-level memory allocator underlying `malloc()`. (`malloc()` allocates only memory between HeapBase and HeapTop; a call to `__rt_alloc()` attempts to move HeapTop: cf Unix `sbrk()`). `__rt_alloc` should try to allocate a block of sensible size >= `minwords`. If this is not available, and if `__osdep_heapsupport_extend` is defined, it should call that 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 initialise 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 initialisation (if `malloc` is linked into the image). The use of `__rt_malloc` ensures that allocations made before `malloc` is initialised succeed, 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 initialise to some primitive memory-freeing function. (see `__rt_malloc` above).

15.6.5 Miscellaneous

`void __osdep_traphandlers_init(void)`

This arranges to catch processor aborts (and pass 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 number of bytes acquired, and 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`.

16 Software Floating Point

This chapter describes the ARM software floating point arithmetic library.

Software Floating Point

16.1 Introduction

Floating point arithmetic on the ARM has traditionally been done using the floating point extensions to the ARM instruction set. Systems use either a hardware floating point unit (the FPA), or a software emulator (the FPE) which traps calls to the FPA. The system allows code to take advantage of a hardware unit if one is available, without the need for recompilation.

Floating point variables are held in registers on the (emulated) FPA, and co-processor instructions are used to manipulate these variables. However the extra cost of intercepting, decoding and emulating the FPA instruction set is quite high.

In many applications (eg. embedded control) the flexibility offered by this approach is not required. In such situations the use of a software floating point library is a considerable advantage.

For more information on floating point, refer to the FPA datasheet, and IEEE754 (the IEEE standard for binary floating point arithmetic).

16.2 The ARM Floating Point Library

The ARM software floating point library provides a set of functions that the C compiler uses in place of the FPA instructions (eg. `_dadd` to add two doubles). Although based on the FPE, the library removes much of the overhead of emulating the FPA.

This approach has a number of advantages over the FPE:

- Significantly faster code.

  By avoiding the decoding and emulation of the FPA instructions, the floating point library typically achieves twice the floating point performance of the FPE.
- Smaller code.

  Although the executable for a given program will be larger since all the floating point code is linked to it, the actual memory used in a system should be less because there is no need to include the FPE. For example the `linpack` program increases in size from 24Kb to 34Kb, but no longer needs 26Kb of FPE.
- No need to port the FPE to your target environment.

  The FPE must be modified for use in a new environment because it effectively forms part of the operating system. It requires some dedicated workspace which must be allocated in the target memory map. Multi-tasking environments must preserve the floating point context between task switches. There is no need for any porting if the software library is used. This reduces development time significantly.

The main disadvantage is that code compiled using the library will not take advantage of a hardware floating point accelerator.

16.3 Usage

There is a new APCS option to control which floating point mechanism is used by `armcc`:

`-apcs /softfp`	generate calls to ARM software floating point library (default)
`-apcs /hardfp`	generate in-line ARM floating point instructions

Note that the compiler warns if you use `softfp` in conjunction with the following since these options only apply to a `hardfp` system.

```
    -apcs /fpe3
    -apcs /fpe2
-apcs /fpregargs
```

Thumb: The `tcc` always uses software floating point.

16.4 Interworking Between hardfp and softfp Systems

Functions that return a floating point type under software floating point use the ARM's integer registers (returning a double in r0 and r1, and a float in r0). Under the FPE results are returned in the floating point register f0. Hence the two are not compatible.

You should not need to mix ARM floating point instructions and calls to the `softfp` library.

16.5 Calling the Floating Point Library from Assembler

The software floating point library provides a number of functions for basic floating point operations. IEEE double precision (`double`) values are passed in pairs of registers, and single precision (`float`) numbers in a single register.

For example `_dadd` is the function to add two double precision numbers and return the result.

It can be considered as having the prototype:

```
extern double _dadd(double, double);
```

that is, the two numbers to be added are passed in R0/R1 and R2/R3, and the result is returned in R0/R1.

Similarly the function `_fadd` (single precision add) has the two arguments passed in R0 and R1, and the result returned in R0.

The complete set of functions are given in ➲*Table 16-1: Library functions*:

Function	Operation	Arg1 (type)	Arg2 (type)	Result (type)
`_dadd`	`A+B`	`R0/R1 (double)`	`R2/R3 (double)`	`R0/R1 (double)`
`_dsub`	`A-B`	`R0/R1 (double)`	`R2/R3 (double)`	`R0/R1 (double)`
`_drsb`	`B-A`	`R0/R1 (double)`	`R2/R3 (double)`	`R0/R1 (double)`
`_dmul`	`A*B`	`R0/R1 (double)`	`R2/R3 (double)`	`R0/R1 (double)`
`_ddiv`	`A/B`	`R0/R1 (double)`	`R2/R3 (double)`	`R0/R1 (double)`
`_drdv`	`B/A`	`R0/R1 (double)`	`R2/R3 (double)`	`R0/R1 (double)`
`_dneg`	`-A`	`R0/R1 (double)`		`R0/R1 (double)`
`_fadd`	`A+B`	`R0 (float)`	`R1 (float)`	`R0 (float)`
`_fsub`	`A-B`	`R0 (float)`	`R1 (float)`	`R0 (float)`
`_frsb`	`B-A`	`R0 (float)`	`R1 (float)`	`R0 (float)`
`_fmul`	`A*B`	`R0 (float)`	`R1 (float)`	`R0 (float)`
`_fdiv`	`A/B`	`R0 (float)`	`R1 (float)`	`R0 (float)`
`_frdv`	`B/A`	`R0 (float)`	`R1 (float)`	`R0 (float)`
`_fneg`	`-A`	`R0 (float)`		`R0 (float)`
`_dgr`	`A>B`	`R0/R1 (double)`	`R2/R3 (double)`	`R0 (boolean)`
`_dgeq`	`A>=B`	`R0/R1 (double)`	`R2/R3 (double)`	`R0 (boolean)`
`_dls`	`A<B`	`R0/R1 (double)`	`R2/R3 (double)`	`R0 (boolean)`
`_dleq`	`A<=B`	`R0/R1 (double)`	`R2/R3 (double)`	`R0 (boolean)`
`_dneq`	`A!=B`	`R0/R1 (double)`	`R2/R3 (double)`	`R0 (boolean)`
`_deq`	`A==B`	`R0/R1 (double)`	`R2/R3 (double)`	`R0 (boolean)`
`_fgr`	`A>B`	`R0 (float)`	`R1 (float)`	`R0 (boolean)`
`_fgeq`	`A>=B`	`R0 (float)`	`R1 (float)`	`R0 (boolean)`

Table 16-1: Library functions

Function	Operation	Arg1 (type)	Arg2 (type)	Result (type)
`_fls`	`A<B`	`R0 (float)`	`R1 (float)`	`R0 (boolean)`
`_fleq`	`A<=B`	`R0 (float)`	`R1 (float)`	`R0 (boolean)`
`_fneq`	`A!=B`	`R0 (float)`	`R1 (float)`	`R0 (boolean)`
`_feq`	`A==B`	`R0 (float)`	`R1 (float)`	`R0 (boolean)`
`_dflt`	`(double)A`	`R0 (int)`		`R0/R1 (double)`
`_dfltu`	`(double)A`	`R0 (unsigned)`		`R0/R1 (double)`
`_dfix`	`(int)A`	`R0/R1 (double)`		`R0 (int)`
`_dfixu`	`(unsigned)A`	`R0/R1 (double)`		`R0 (unsigned)`
`_fflt`	`(float)A`	`R0 (int)`		`R0 (float)`
`_ffltu`	`(float)A`	`R0 (unsigned)`		`R0 (float)`
`_ffix`	`(int)A`	`R0 (float)`		`R0 (int)`
`_ffixu`	`(unsigned)A`	`R0 (float)`		`R0 (unsigned)`
`_d2f`	`(double)A`	`R0 (float)`		`R0/R1 (double)`
`_f2d`	`(float)A`	`R0/R1 (double)`		`R0 (float)`

Table 16-1: Library functions

In addition the library provides some functions for dealing with extended precision values, but these are not documented.

Thumb: In the Thumb software floating point library the thumb entry points are predefined with `__16` (eg. `__16_dadd`). This is to allow both ARM and Thumb versions of floating point functions to be present when interworking ARM and Thumb.

16.6 Controlling Floating Point Exceptions from C

Both the `hardfp` and `softfp` modes provide a function (`__fp_status`) for setting and reading the status of either the FPE/FPA or the floating point library.

The following is an extract from `stdlib.h`:

```
extern unsigned int __fp_status(unsigned int /* mask */,
unsigned int /*flags*/);
#define __fpsr_IXE  0x100000      Inexact exception trap enable bit
#define __fpsr_UFE  0x80000       Underflow exception trap enable bit
#define __fpsr_OFE  0x40000       Overflow exception trap enable bit
#define __fpsr_DZE  0x20000       Divide by zero exception trap enable bit
#define __fpsr_IOE  0x10000       Invalid operation exception trap enable bit

#define __fpsr_IXC  0x10          Inexact exception flag bit
#define __fpsr_UFC  0x8           Underflow exception flag bit
#define __fpsr_OFC  0x4           Overflow exception flag bit
#define __fpsr_DZC  0x2           Divide by zero exception flag bit
#define __fpsr_IOC  0x1           Invalid operation exception flag bit
```

`mask` and `flags` are bit-fields which correspond directly to the floating point status register (FPSR) in the FPE/FPA and the software floating point library.

The function `__fp_status` returns the current value of the status register, and also sets the writable bits of the word (the exception control and flag bytes) to:

```
new = (old & ~mask) ^ flags;
```

Four different operations can be performed on each status register bit, determined by the respective bits in `mask` and `flags`:

mask bit	flags bit	Effect
0	0	no effect
0	1	toggle bit in status register
1	0	clear bit in status register
1	1	set bit in status register

Table 16-2: Status register bit operations

The `__fp_status` function always returns the current value of the status register, before any changes are applied.

Initially all exceptions are enabled, and no flags are set.

16.6.1 Examples

```
status = __fp_status(0,0);
/* reads the status register, does not change it */

__fp_status(__fpsr_DZE,0);
/* disable divide-by-zero exception trap */

inexact = __fp_status(__fpsr_OFC,0) & __fpsr_OFC;
/* read (and clear) overflow exception flag bit */

/* Report the type of floating point system being used. */
switch (flags=(__fp_status(0,0)>>24))
    {
       case 0x0: case 0x1:
              printf("Software emulation\n");
              break;
       case 0x40:
              printf("Software library\n");
              break;
       case 0x80: case 0x81:
              printf("Hardware\n");
              break;
       default:
              printf("Unknown ");
              if (flags & (1<<7))
                       printf("hardware\n");
              else
                       printf("software %s\n",
                               flags & (1<<6) ? "library"
                               : "emulation");
              break;
       }
```

16.6.2 System ID byte

Bits 31:24 contain a system ID byte. The currently defined values are:

0x00	Pre-FPA floating point emulator
0x01	FPA compatible floating point emulator
0x40	Floating point library

0x80	FPPC (obsolete)
0x81	FPA10 (with FPSC module)

The top bit (bit 31) is used to distinguish between hardware and software systems, and bit 30 is used to distinguish between software emulators and libraries.

16.6.3 Exception trap enable byte

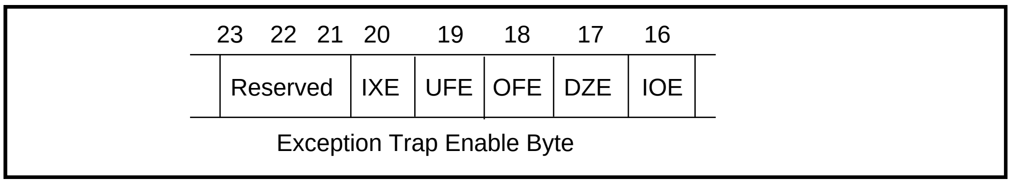
Each bit of the exception trap enable byte corresponds to one type of floating point exception.

Bits 23:16 control the enabling of exceptions on floating point errors:

bits 23:21		Reserved
bit 20	IXE	Inexact exception enable*
bit 19	UFE	Underflow exception enable*
bit 18	OFE	Overflow exception enable
bit 17	DZE	Divide by zero exception enable
bit 16	IOE	Invalid operation exception enable

A set bit causes the system to take an exception trap if an error occurs. Otherwise a bit is set in the cumulative exception flags (see below) and the IEEE defined result is returned.

Note: *The current floating point library will never produce those exceptions marked with a *.*



Exception Trap Enable Byte

A bit in the cumulative exception flags byte is set as a result of executing a floating point instruction only if the corresponding bit *is not* set in the exception trap enable byte. If the corresponding bit in the exception trap enable byte *is* set, a runtime error will occur (`SIGFPE` will be raised in a C environment).

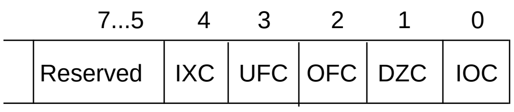
16.6.4 System control byte

This byte is not used on the floating point library system. Refer to the FPA datasheet for details of its meaning under FPA and FPE systems.

In particular, the NaN exception control bit (bit 9) is not yet supported by the floating point library, but may be in a future version.

16.6.5 Exception flags byte

Bits 7:0 contain flags for whether each exception has occurred in the same order as the Exception Trap Enable Byte. Exceptions occur as defined by IEEE 754.



Cumulative Exception Flags Byte

16.7 Formats

`int` and `unsigned`	32-bit integer quantities.
`boolean`	Either 0 (False) or 1 (True).
`float`	IEEE single precision floating point number.

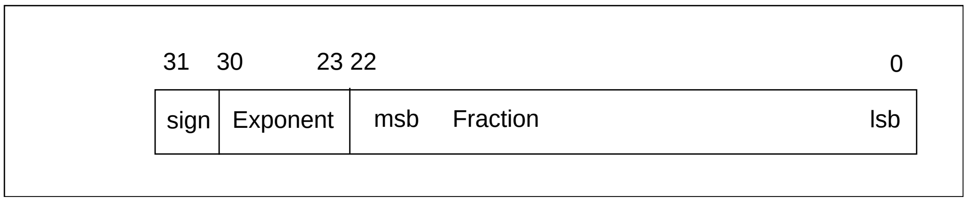


Figure 16-1: IEEE single precision

`double`	IEEE double precision floating point number.

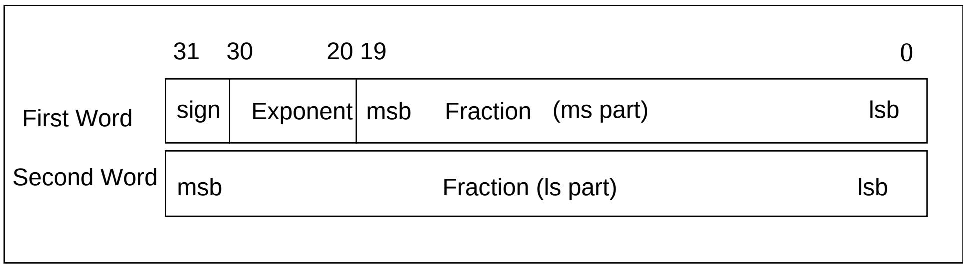


Figure 16-2: IEEE double precision

17 Demon

This chapter describes the ARM debug monitor, Demon.

Demon

17.1 Introduction

The ARM debug monitor, `demon`, provides a common low-level programming, C library and debugging environment across the ARM emulator, linked with *the symbolic debugger.*

This chapter describes the common programming and debugging environment supported by implementations of demon. For implementation details refer to the following:

- For the ARMulator model of demon, see `armos.h` and `armos.c` in the `armul` directory of this release.
- For the implementation used in the PIE, see the *PIE User Manual*.
- To see how the functions of the debug monitor are used to implement a semi-hosted C library, see the `semi` subdirectory of the `clib` directory in this release.

17.2 Target Memory Map

An initial memory map is defined as follows:

Address	Description
`0x00000`	CPU Reset Vector
`0x00004`	CPU Undefined Instruction Vector
`0x00008`	CPU Software Interrupt Vector
`0x0000c`	CPU Prefetch Abort Vector
`0x00010`	CPU Data Abort Vector
`0x00014`	CPU Address Exception Vector
`0x00018`	CPU Interrupt Vector
`0x0001c`	CPU Fast Interrupt Vector
`0x00020`	~1 KBytes for FIQ routine and FIQ mode stack
`0x00400`	256 bytes for IRQ mode stack
`0x00500`	256 bytes for Undefined mode stack
`0x00600`	256 bytes for Abort mode stack
`0x00700`	256 bytes for SVC mode stack
`0x00800`	Debug monitor Private Workspace
`0x01000`	Free for user supplied Debug Monitor
`0x02000`	Floating Point Emulator space
`0x08000`	Application space
`Top of memory`	an implementation-dependent address returned by the debug monitor SWI `SWI_Getenv`. For the PIE and ARMulator variant it has the value 0x80000 (512 KBytes). The C library support uses the top of memory address for the base of the user mode stack. All stacks grow towards address zero.

Table 17-1: Initial memory map

17.3 Standard Monitor SWIs

Demon implements a number of useful SWIs:

`SWI_WriteC (SWI 0)`

Write a byte, passed in register 0, to the debug channel. When executed under the symbolic debugger, the character will appear on the display device connected to the debugger.

`SWI_Write0 (SWI 2)`

Write the null-terminated string, pointed to by register 0, to the debug channel. When executed under the symbolic debugger, the characters will appear on the display device connected to the debugger.

`SWI_ReadC (SWI 4)`

Read a byte from the debug channel, returning it in register 0. The read is notionally from the keyboard attached to the debugger.

`SWI_CLI (SWI 5)`
This SWI is not available in the PC/DOS release.

Pass the string, pointed to by register 0, to the host's command line interpreter.

`SWI_GetEnv (SWI 0x10)`

Returns in register 0 the address of the command-line string used to invoke the program, and in register 1 the highest available address in user memory.

`SWI_Exit (SWI 0x11)`

Halt emulation. This is the way a program exits cleanly, returning control to the debugger.

`SWI_EnterOS (SWI 0x16)`

Put the processor into supervisor mode. For a 32-bit demon, `SVC32` is entered. If the demon was built with `CONFIG=26`, `SVC26` is entered.

`SWI_GetErrno (SWI 0x60)`

Return, 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 `SWI_Remove, SWI_Rename, SWI_Open, SWI_Close, SWI_Read, SWI_Write, SWI_Seek`, etc. Whether or not, and to what value `errno` is set is completely host-specific, except where the ANSI C standard defines the behaviour.

`SWI_Clock (SWI 0x61)`

Return, in r0, the number of centi-seconds since the support code began execution. In general, only the difference between successive calls to `SWI_Clock`, can be meaningful.

`SWI_Time (SWI 0x63)`

Return, in r0, the number of seconds since the beginning of 1970 (the Unix time origin).

`SWI_Remove (SWI 0x64)`

Delete (remove, un-link, wipe, destroy) the file named by the nul-terminated string addressed by r0. Return (in r0) a zero if the removal succeeds, or a non-zero, host-specific error code if it fails.

`SWI_Rename (SWI 0x65)`

r0 and r1 address NUL-terminated strings, the *old-name* and *new-name*, respectively. If the rename succeeds, zero is returned in r0; otherwise, a non-zero, host-specific error code is returned.

`SWI_Open (SWI 0x66)`

r0 addresses a NUL-terminated string containing a file or device name; r1 is a small integer specifying the file-opening mode, as shown in *Table 17-2: File-opening mode.* If the open succeeds, a non-zero handle is returned in r0, which can be quoted to `SWI_Close, SWI_Read, SWI_Write, SWI_Seek, SWI_Flen and SWI_IsTTY.` Nothing else may be asserted about the value of the handle. If the open fails, the value 0 is returned in r0.

r1 value	**ANSI C fopen() mode**
`0`	`"r"`
`1`	`"rb"`
`2`	`"r+"`
`3`	`"r+b"`
`4`	`"w"`
`5`	`"wb"`
`6`	`"w+"`
`7`	`"w+b"`
`8`	`"a"`
`9`	`"ab"`
`10`	`"a+"`
`11`	`"a+b"`

Table 17-2: File-opening mode

`SWI_Close (SWI 0x68)`

On entry, r0 must be a handle for an open file, previously returned by `SWI_Open`. If the close succeeds, zero is returned in r0; otherwise, a non-zero value is returned.

`SWI_Write (SWI 0x69)`

On entry, r0 must contain a handle for a previously opened file; r1 points to a buffer in the callee; and r2 contains the number of bytes to be written from the buffer to the file. `SWI_Write` returns, in r0, the number of bytes *not* written (and so indicates success with a zero return value).

`SWI_Read (SWI 0x6a)`

On entry, r0 must contain a handle for a previously opened file or device; r1 points to a buffer in the callee; and r2 contains the number of bytes to be read from the file into the buffer. `SWI_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 (`SWI_IsTTY` returns non-zero for this handle), then a non-zero return from `SWI_Read` indicates that the line read did not fill the buffer.

`SWI_Seek (SWI 0x6b)`

On entry, r0 must contain a handle for a seekable file object, and r1 the absolute byte position to be sought to. If the request can be honoured then `SWI_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.

`SWI_Flen (SWI 0x6c)`

On entry, r0 contains 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.

`SWI_IsTTY (SWI 0x6e)`

On entry, r0 must contain a handle for a previously opened file or device object. On exit, r0 contains 1 if the handle identifies an interactive device; otherwise r0 contains 0.

`SWI_TmpNam (SWI 0x6f)`

On entry, r0 points to a buffer and r1 contains the length of the buffer (r1 should be at least the value of `L_tmpnam` on the host system). On successful return, r0 points to the buffer which contains a host temporary file name. If the request cannot be satisfied (eg. because the buffer is too small) then 0 is returned in r0.

`SWI_InstallHandler (SWI 0x70)`

`SWI_InstallHandler` installs a handler for a hardware exception. On entry, r0 contains the exception number (see *Table 17-3: Hardware exception handling* on page 17-7); r1 contains a value to pass to the handler when it is eventually invoked; and r2 contains the address of the handler. On return, r2 contains the address of the previous handler and r1 contains its argument.

When the exception occurs, the handler is entered in the appropriate non-user mode, with r10 holding a value dependent on the exception type, and r11 holding the handler argument (as passed to InstallHandler in r1). r10, r11, r12 and r14 (for the processor mode in which the handler is entered) are saved on the stack of the mode in which the handler is entered; all other registers are as at the time of the exception. (Any effects of the instruction causing the exception have been unwound, and the saved r14 points at the instruction that failed, rather than one or two words ahead.) If the handler returns, the exception will be passed to the debugger (regardless of the value of the debugger variable $vector_catch).

No.	Exception	Mode	r10 value
0	branch through zero	svc32	-
1	undefined instruction	undef32	-
2	swi	svc32	swi number
3	prefetch abort	abort32	-
4	data abort	abort32	-
5	address exception	svc32	-
6	IRQ	irq32	-
7	FIQ	fiq32	-
8	Error	svc32	error pointer

Table 17-3: Hardware exception handling

`SWI_GenerateError (SWI 0x71)`

On entry, r0 points to an error block (containing a 32-bit error number, followed by a zero-terminated error string); r1 points to a 17-word block containing the values of the ARM CPU registers at the instant the error occurred (the 17th word contains the PSR).

`SWI_GenerateError` calls the (software) error vector: if bit 8 of `$vector_catch` is set this causes the debugger to be entered directly; otherwise, the installed error handler is called (see `SWI_InstallHandler`).

17.4 The Implementation of Demon for the PIE Card

The debug monitor is split into two parts, called Level 0 and Level 1. The code in Level 1 is replaceable, allowing new targets to be debugged using the same standard interface. The Remote Debug Protocol (RDP) is implemented between the debugger and the debuggee. See *22.2 ARM Remote Debug Protocol* on page 22-21 for a complete description. Quite simply, the RDP defines a protocol that can communicate with a debugger and a debug monitor. Level 0 is responsible for recognising incoming RDP messages from the debug channel, and dispatching them to Level 1 of the debug monitor. Level 0 also drives the debug channel, at the request of Level 1.

Level 1 handles RDP messages from the debugger, and generates RDP messages to support the actions of the application using it. The Level 1 code implements a number of SWI instructions to allow an application access to high level functions. See *17.3 Standard Monitor SWIs* on page 17-4.

The interface between Level 0 and Level 1 is via a number of entry addresses situated at the beginning the Level 0 code. Their layout is as follows:

Offset	
`0x00000`	Reset Instruction
`0x00004`	Address of Reset routine
`0x00008`	Address of InstallRDP routine
`0x0000c`	Address of ResetChannel routine
`0x00010`	Address of ChannelSpeed routine
`0x00014`	Address of GetByte routine
`0x00018`	Address of PutByte routine
`0x0001c`	Address of ReadTimer routine
`0x00020`	Address of SetLED routine

The first two words are mapped into address 0 and 4 when the CPU is reset to provide the initial code entry.

17.4.1 Debug channel routines

`InstallRDP`	is used by the Level 1 code to register a handler of all RDP messages. The address of the handler should be passed in register 0, and the address of the previous handler will be returned in register 0. Level 1 code that does not wish to handle all RDP messages may pass unhandled messages to the previous handler. The RDP handler is entered in FIQ mode, with interrupts disabled and the RDP message number in register 0. Exit from the handler by loading the program counter from the stack with an instruction such as: `LDMFD sp!, {pc}` All register values have been saved before entry to the handler, and will be restored after exit. Once the Level 1 RDP handler has been entered, it may read successive bytes from the debug channel using the `GetByte` routine, and send replies using the `PutByte` routine. The Level 1 handler may also formulate RDP messages to support application code that it has been called by, and these too are sent using `PutByte`.
`ResetChannel`	can be used to reset the debug channel driver, should an error be detected by the Level 1 Code.
`ChannelSpeed`	is used to change the speed of the debug channel in an implementation designed fashion. Currently the PIE board supports baud rates of 9600, 19200 and 38400 bits per second over its serial channel. These speeds can be selected by sending the values 1, 2 or 3 respectively to the `ChannelSpeed` routine. A value of 0 selects the default (power on reset) value, which for the PIE is 9600 bps. The meaning of all other values is undefined.
`GetByte`	gets bytes to and from the debug channel, as described above.
`PutByte`	puts bytes to and from the debug channel, as described above.
`ReadTimer`	returns in register 0 a centi-second count from an on board timer. If an on-board timer is not available, this routine will always return 0xFFFFFFFF (-1).
`SetLED`	allows the setting and clearing of an on-board Light Emitting Diode (LED). Register 0 dictates the action required: 0 turns the LED off; any other values turn it on.

18 EmbeddedICE

This chapter describes aspects of EmbeddedICE.

18.1 The Effect of EmbeddedICE on the Debuggee

EmbeddedICE has been designed to allow debugging via the JTAG port to be as non-intrusive as possible:

- The debuggee needs no special hardware to support debugging (the ICEbreaker and the JTAG TAP controller is all that is required).
- No memory in the debuggee system need be set aside for debugging, and no special software need be incorporated to allow debugging.
- Execution of the debuggee should only be halted when a breakpoint or watchpoint has been hit or the user requests that the debuggee be halted.

Although EmbeddedICE is generally non-intrusive, there are two exceptions:

- When armsd is started up it attempts to find out the state of the debugee. To do this, it halts the debuggee and inspects the state of the ARM registers. This, however, can be considered non-intrusive if the debugging session is started once armsd has been started.
- Watchpoints on structures or arrays larger than one word may cause the debuggee to halt execution when writes occur close to the watchpointed area. EmbeddedICE will restart execution transparently to the user, but this may still cause problems if the application is real time. For more information see ➲*7.3.1 Extensions to armsd for EmbeddedICE* on page 7-4.

18.2 Vector Breakpoints

When you start execution (using `go` or `step`), EmbeddedICE puts into place any breakpoints specified by the `armsd` internal variable `$vector_catch`. This means that when you start executing, user breakpoints and watchpoints may have to be downgraded from hardware ones to software ones without any warning being given. See the Software Development Toolkit documentation for more information.

18.3 Configuration Data

Building a configuration data file is not generally required, so you may not need to read this section.

EmbeddedICE has built in configuration data for ARM70DM. However, if for any reason different configuration data needs to be supplied to EmbeddedICE you can

- use `armsd` commands
- use `armsd` invocation options.

The syntax of these is described in ➲*7.3.1 Extensions to armsd for EmbeddedICE* on page 7-4.

A configuration file consists of a number of configuration blocks simply concatenated. Each block contains data for a particular system. You can load a configuration file using `loadconfig`. You can then select the desired configuration block using `selectconfig`.

Building a configuration block is relatively simple, and is done using `armcc` and `armlink`. The configuration block built into EmbeddedICE for ARM70DM has been supplied in a form which can be modified if necessary. The `iceconf.h` file contains well commented descriptions of all the data structures needed in a configuration block so, together with the arm70dm example configuration file, should allow new configuration data blocks to be created.

A readme file is included with simple instructions to build this configuration block It can be found in the `config` directory.

18.4 Accessing the EmbeddedICE Macrocell Directly

No `armsd` commands have been added to allow you to manipulate ICEbreaker registers directly. Instead you can use the commands which display and set coprocessor registers, with coprocessor number 0 specified. Coprocessor 0 is defined to never be implemented, so this cannot clash with user or ARM developed coprocessors.

For example, the command `cregisters 0` will display the contents of a number of registers, which are in fact ICEbreaker's registers. The correspondence between coprocessor 0 registers displayed and ICEbreaker registers is simple. The `register address` field in the ICEbreaker scan chain is precisely the register number. See your ARM data sheet for more information about ICEbreaker.

You may read ICEbreaker registers freely in this manner, but writing them needs some more care, as EmbeddedICE also makes use of ICEbreaker registers to set up breakpoints and watchpoints. Thus when you write an ICEbreaker register (eg. `cwrite 0 20 0x44`), EmbeddedICE checks to see if the breakpoint which that register is a part of is in use. If it is, EmbeddedICE will attempt to free it (by degrading hardware breakpoints to software breakpoints), and then set a lock on that breakpoint so that EmbeddedICE makes no further attempt to use it.

It is possible to see which breakpoints have been locked in this way by printing the value of `$icebreaker_lockedpoints`. This `armsd` internal variable can also be set to unlock breakpoints. In the ARM70DM the breakpoints are numbered 1 and 2, and bits 1 and 2 in `$icebreaker_lockedpoints` indicate their status.

If a breakpoint or watchpoint which has been set up in this way gets taken, EmbeddedICE will not know why execution has stopped (it wasn't due to one of the break/watch points it knows about), and so will halt debuggee execution reporting `Unknown watchpoint`.

Take care when writing ICEbreaker registers 0 and 1, the control and status registers, as EmbeddedICE uses these to perform many of its operations. Thus if these are written at all, they should always be returned to their original values afterwards.

Note also that requests to armsd to read or write ICEbreaker registers do not necessarily cuase the registers to be read or written directly. This is because, in the interests of efficiency, the B/I S/W caches the contents of the ICEbreaker registers, only updating changed registers before execution of the debuggee is resumed.

18.5 Floating Point and Other Coprocessors

By default EmbeddedICE assumes that there are no coprocessors attached to the debuggee. If in fact there are coprocessors attached, then a suitable `coproc` command should be issued to `armsd`.

19 ARM Procedure Call Standard

This chapter describes the ARM procedure call standard.

ARM Procedure Call Standard

19.1 Introduction

The *ARM Procedure Call Standard* (*APCS*) is a set of rules which regulate and facilitate calls between separately compiled or assembled program fragments.

The APCS defines:

- constraints on the use of registers
- stack conventions
- 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 (see ➲*6.11 ARM Shared Library Format* on page 6-18 for details)

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 ➲*19.3 APCS Variants* on page 19-11 for details of the variants in the family. Implementors of run-time systems, operating systems, embedded control monitors, etc., must choose the variant(s) most appropriate to their requirements.

Naturally, there can be 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.

19.1.1 Design criteria

Throughout its history, the APCS has compromised between *fastest*, *smallest* and *easiest to use*.

The criteria we have considered to be important are:

- Function calls should be fast and it should be easy for compilers to optimise 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 writeable 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 run-time error diagnosis.

Overall, we have tended to rank compact code and a clear definition most highly, with simplicity and ease of use ahead of performance in matters of fine detail where the impact on performance is small.

19.2 The ARM Procedure Call Standard

This section defines the ARM Procedure Call Standard. Explanatory text, not itself part of the standard, is given in parentheses.

Program fragments

(The following explanation may help you to understand the APCS but is not itself part of the APCS. If an explanation appears to conflict with the standard, the standard should be considered definitive and the narrative merely an indication of intent.)

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 (and if so, when) program fragments for a particular ARM-based environment are required to conform strictly to the APCS, is part of the definition of that environment.

19.2.1 Register names

The ARM has 15 visible general registers, a program counter register and 8 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 19-1: ACPS registers* on page 19-4.

19.2.2 General registers

Register Number	APCS Name	APCS Role
0	a1	argument 1 / integer result / scratch register
1	a2	argument 2 / scratch register
2	a3	argument 3 / scratch register
3	a4	argument 4 / scratch register
4	v1	register variable
5	v2	register variable
6	v3	register variable
7	v4	register variable
8	v5	register variable
9	sb/v6	static base / register variable
10	sl/v7	stack limit / stack chunk handle / reg. variable
11	fp	frame pointer
12	ip	scratch register / new-sb in inter-link-unit calls
13	sp	lower end of current stack frame
14	lr	link address / scratch register
15	pc	program counter

Table 19-1: ACPS registers

Notes: The 16 integer registers are divided into 3 sets:

- 4 argument registers which can also be used as scratch registers or as caller-saved register variables;
- 5 callee-saved registers, conventionally used as register variables;
- 7 registers which have a dedicated role, at least some of the time, in at least one variant of APCS-3 (see *19.3 APCS Variants* on page 19-11.)

The 5 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.

19.2.3 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 2 words; an extended precision value occupies 3 words, as does an internal precision value.)

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 19-2: Floating point registers

(The floating point (FP) registers are divided into two sets, analogous to the subsets a1-a4 and v1-v5/v7 of the general registers:

- Registers f0-f3 need not be preserved by called functions; f0 is the FP result register and f0-f3 may hold the first four FP arguments: see ➲*19.2.8 Data representation and argument passing* on page 19-10 and ➲*19.3 APCS Variants* on page 19-11.
- Registers f4-f7, the so called 'variable' registers, preserved by callees.)

19.2.4 The stack

The stack is a singly-linked list of *activation records*, linked through a *stack backtrace data structure* (see ➲*19.2.5 The stack backtrace data structure* on page 19-7), stored at the high-address end of each activation record.

- The stack must be readable and writeable 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 ➲*19.3 APCS Variants* on page 19-11).

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 stack chunk limit: The stack chunk limit is said to be *implicit* if chunk overflow is detected and handled by the execution environment: otherwise it is *explicit*.

If the stack chunk limit is implicit, sl may be used as v7, an additional callee-saved variable register.

If the conditions of the remainder of this subsection hold at all instants of execution, then the program conforms strictly to the APCS; otherwise, if they hold at and during external (inter-compilation-unit-visible) function calls, 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, 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.)

19.2.5 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 has the format:

```
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
```

This shows between four and twenty-seven 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.

19.2.6 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.)

Function exit

Value	**Restored to**
`return link value`	pc
`return sp value`	sp
`return fp value`	fp

APCS variants

APCS variant	**Save code pointer**
32-bit PC variant	Must point twelve bytes beyond the start of the sequence of instructions that created the stack backtrace data structure.
26-bit PC variant	When cleared of PSR and mode bits, must point twelve bytes beyond the start of the sequence of instructions that created the stack backtrace data structure.

19.2.7 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 ➲*19.2.5 The stack backtrace data structure* on page 19-7)
  (In 26-bit variants of the APCS, lr contains the PC + PSR value to restore to pc on exit from the function: see ➲*19.3 APCS Variants* on page 19-11.)
- 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 ➲*19.2.4 The stack* on page 19-6
- fp contains 0 or points to the most recently created stack backtrace structure: see ➲*19.2.5 The stack backtrace data structure* on page 19-7

- the space between sp and the stack chunk limit is readable and writeable memory which the called function can use as temporary workspace and overwrite with any values before the function returns: see *19.2.4 The stack* on page 19-6
- arguments are marshalled as described in *19.2.8 Data representation and argument passing* on page 19-10

A re-entrant target function (see *19.3 APCS Variants* on page 19-11) 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: usually they will be 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.
See *6.11 ARM Shared Library Format* on page 6-18 for details of support for re-entrant code and shared libraries.

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 sharable, sb may be used as v6, an additional variable register—see *Table 19-1: ACPS registers* on page 19-4).

19.2.8 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 1, 2 or 3 words)

A callee may corrupt any of its arguments, howsoever 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. Indeed, there is no formal reason why two ARM-targeted implementations of the same language should not use different mappings and, hence, not support cross-calling. Obviously, it would be very unhelpful to stand by this formal position so implementors are encouraged to adopt not just the letter of the APCS but also the natural mappings of source language objects into argument words. Guidance about this is given in ➲*19.4 C Language Calling Conventions* on page 19-14.)

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 ➲*19.3 APCS Variants* on page 19-11), the first 4 floating point arguments (or fewer if the number of floating point arguments is less than 4) are in machine registers f0-f3
- the first 4 remaining argument words (or fewer if there are fewer than 4 argument words remaining in the argument list) are in machine registers a1-a4
- the remainder of the argument list (if any) are in memory, at the location addressed by sp and higher-addressed words thereafter

A floating point value not passed in a floating point register is treated as 1, 2 or 3 integer values, as appropriate to its precision.

19.2.9 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 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 *19.4 C Language Calling Conventions* on page 19-14)
- 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 saves* 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 *19.3 APCS Variants* on page 19-11.)

19.3 APCS Variants

There are 2 x 2 x 2 x 2 = 16 APCS variants, derived from four independent choices. In each case, code conforming to one variant is not compatible with code conforming to the other. The only true user-level choice is between re-entrant versus non-re-entrant variants. Since the alternatives are compatible, each may be used where appropriate.

1. **32-bit PC vs 26-bit PC**. This is fixed by your ARM CPU.
2. **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.
3. **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.

   b) the floating-point instruction set is supported by hardware or floating-point operations are dynamically common.

19.3.1 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, and pack the 4 status flags (NZCV) and 2 interrupt-enable flags (IF) into the top 6 bits of r15, and the 2 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 would be 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.

19.3.2 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.

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).

The majority of ARM-based systems are expected to require software stack-limit checking.

19.3.3 Floating point arguments in floating point registers

Historically, many ARM-based systems have made no use of the floating point instruction set, or have used a software emulation of it.

On systems using a slow software emulation and making little use of floating point, there is a small disadvantage to 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.*

19.3.4 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: see *6.11 ARM Shared Library Format* on page 6-18 for further details.

In general, code to be placed in ROM or loaded into a shared library is expected to be re-entrant, while applications are expected not to be re-entrant.

See also *19.4 C Language Calling Conventions* on page 19-14.

19.3.5 APCS-2 compatibility

(APCS-2— the second definition of The ARM Procedure Call Standard—is recorded in Technical Memorandum *PLG-APCS, issue 4.00, 18-Apr-89*, reproduced in the following Acorn publications: *RISC OS Programmer's Reference Manual, vol IV, 1989*, (Acorn part number 0483,023); *ANSI C Release 3, September 1989*, (Acorn part number 0470,101)).

APCS-R (APCS-2 for Acorn's RISC OS) is the following variant of APCS-3:

- 26-bit PC
- explicit stack-limit checking
- no passing of floating point arguments in floating point registers
- non-re-entrant code

with the Acorn-specific constraints on the use of sl noted in APCS-2.

APCS-U (APCS-2 for Acorn's RISCiX) is the following variant of APCS-3:

- 26-bit PC
- implicit stack-limit checking (with sl reserved to Acorn)
- no passing of floating point arguments in floating point registers
- non-re-entrant code

The (in APCS-2) obsolescent APCS-A has no equivalent in APCS-3.

19.4 C Language Calling Conventions

19.4.1 Argument representation

A floating point value occupies 1, 2, or 3 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 ➲*16.7 Formats* on page 16-10.)

The C compiler widens arguments of type float to type double to support inter-working between ANSI C and classic C.

Char, short, pointer and other integral values occupy 1 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).

19.4.2 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 4 FP arguments are loaded into FP registers.

The first 4 of the remaining 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, 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 the passing of 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.

19.4.3 Non-simple value return

A non-simple type is any non-floating point type of size greater than 1 word (including structures containing only floating point fields), and certain 1 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.

struct {int a:8, b:8, c:8, d:8;} and union {int i; char *p;} are both integer-like; struct {char a; char b; char c; char d;} is not.

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, ...);
```

19.5 Function Entry

A complete discussion of function entry is complex; here we cover 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 difficulties with tail continuations. Suppose stacked arguments are unstacked by callers (almost mandatory for variadic callees), then 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, then some of F's stacked arguments can be replaced by G's stacked arguments. However, this can be hard 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 also termed *frameless.*)

19.5.1 Establishing the static base

(See also ➲*6.11.2 The shared library addressing architecture* on page 6-20.)

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-LU veneers needed between independent re-entrant link units, and between re-entrant and non-re-entrant code.

19.5.2 Creating the stack backtrace structure

For non-re-entrant, non-variadic 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 a1-a4 only has to be saved 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-v7 only have to be saved if it 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 stack...
STMFD  sp!, {v1-v6, fp, ip, lr, pc}  ; 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 the `SFMFD`. `SFM` is not supported by the issue-1 floating point instruction set and must be simulated by 4 `STFE` instructions. See section ➲*19.5.3 Saving and restoring floating point registers* on page 19-18.

For re-entrant variadic functions, the requirements are yet more complicated and the sequence becomes less elegant.

19.5.3 Saving and restoring floating point registers

The issue-2 floating point instruction set defines two new instructions for saving and restoring the floating point registers: Store Floating Multiple (`SFM`) and 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` have to be simulated using sequences of `STFE`s and `LDFE`s.

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.

19.5.4 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 run time

The third case does not arise in C, apart from in stack-based implementations of the non-standard, BSD-Unix `alloca()` function. The APCS does not support `alloca()` in a straightforward manner.

In Modula-2, Pascal and other languages there may be arrays created on block entry or passed as *open array arguments,* the size of which is unknown until run time. These are located in the callee's stack frame, and so impact stack limit checking. In practice this adds little complication—see *19.5.7 Stack limit checking (vari-sized frames)* on page 19-21.

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 run-time support function is called (`__rt_stkovf_split_small` or `__rt_stkovf_split_big`).

Any environment that supports explicit stack limit checking must provide these functions, which can do one of the following:

- terminate execution
- extend the existing stack chunk, decrementing sl
- allocate a new stack chunk, resetting sp and sl to point into it, and guaranteeing that an immediate repeat of the limit check will succeed

19.5.5 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 2 instructions and, in general, only 2 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.

19.5.6 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
```

`FrameSizeBound` can be any convenient constant at least as big as the largest frame the function will use.

Notes: *Functions containing nested blocks may use different amounts of stack at different instants during their execution.*

`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.

19.5.7 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 initialised by the called function

Stack layout

The general scheme for stack layout in this case is shown in *Figure 19-1: Stack layout*, below:

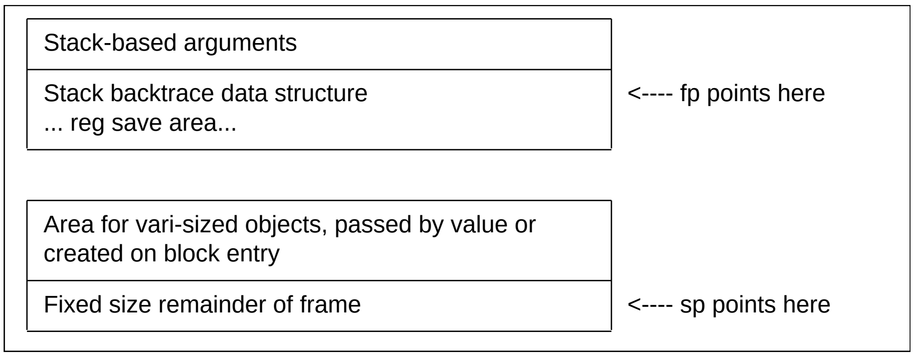

Stack-based arguments
Stack backtrace data structure
... reg save area...
<---- fp points here
Area for vari-sized objects, passed by value or created on block entry
Fixed size remainder of frame
<---- sp points here

Figure 19-1: 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 can't be known until run time, the block entry must include a stack limit check.

19.5.8 Function exit

A great deal of design effort has been devoted to ensuring that 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.

Exit from functions that use no stack and save no floating point registers is simple:

```
MOV    pc, lr
```

(26-bit compatibility demands `MOVS pc, lr` to reinstate the caller’s PSR flags, but this must not be used in 32-bit modes.)

Exit from other functions which save no floating point registers is by:

```
LDMEA  fp, {v1-v5, sb, fp, sp, pc}; as saved
```

Here, it is important that fp points just below the `save code pointer,` as this value is not restored, (`LDMEA` is a pre-decrement multiple load).
(26-bit compatibility demands `LDMEA fp, {regs}^`, to reinstate the caller’s PSR flags, but this must not be used in 32-bit modes.)

The saving and restoring of floating point registers is discussed in ➲*19.5.3 Saving and restoring floating point registers* on page 19-18.

19.5.9 Some examples

This section is not intended to be a general guide to the writing of code generators, but highlights some of the optimisations that are particularly relevant to the ARM and to this standard.

In order to make effective use of the APCS, compilers must compile code a procedure at a time: line at a time compilation 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)
    {  --5->cnt;
        return *(5->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 optimisation, 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 optimisation avoids saving and restoring the call-frame registers and saves 5 instructions (and many cycles-17 S cycles on an uncached ARM with N=2S).

19.6 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 it you only need to save and restore r14 in the calling veneer around the SWI. If a compiler can generate in-line SWIs, then it should also generate code to save and restore r14 in-line around the SWI, unless you know that the code will not be executed in SVC mode.

19.6.1 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 because 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 2nd or 3rd 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` then this action is harmless, as r14 has been corrupted anyway, (and, by design, contained nothing of value when a prefetch abort could occur).

20 Thumb Procedure Call Standard

This chapter describes the Thumb procedure call standard.

Thumb Procedure Call Standard

20.1 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 based strongly on the APCS. If you are unfamiliar with the APCS and its terminology, you will find it helpful to read *Chapter 19, 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 the reduced nature of the Thumb instruction set, which makes implementing the full versatility of the APCS difficult.

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.
- Multiple instantiation (calling the same entry point with different sets of static data).

  Multiple instantiation can still be implemented at a user level, by placing all the variables that need to be multiply instantiated in a struct and passing each function a pointer to the struct.
- Direct floating point support.

  Thumb cannot 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.

20.2 Register Names

The Thumb subset has eight visible general purpose registers (R0-R7) plus a stack pointer (SP), link register (LR) and program counter (PC). In addition, the Thumb subset can access all of the ARM registers singly via a set of special instructions: see *Chapter 5, Thumb Instruction Set* for details.

In the context of the TPCS, each of the Thumb registers has a special name and function: these are summarized in *Table 20-1: TPCS registers*, below.

Register Number	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 20-1: TPCS registers

20.3 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 possibly 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.

20.3.1 The stack backtrace data structure

The value in fp is zero, or points to a list of stack backtrace data structures which partially describe the sequence of outstanding function calls.

The stack backtrace data structure has the following format:

```
save code pointer   [fp]       <-fp points to here
return link value   [fp, #-4]
return sp value     [fp, #-8]
return fp value     [fp, #-12]
[saved v4 value]
[saved v3 value]
[saved v2 value]
[saved v1 value]
[saved a4 value]
[saved a3 value}
[saved a2 value]
[saved a1 value]
```

20.3.2 Function invocations and backtrace structures

The `return fp value` is either 0 or a pointer to a stack backtrace data structure created by an ancestor of the function invocation that created the backtrace structure pointed to by fp. No more recent ancestor will have created a backtrace structure.

Note: *This is not necessarily the most recent ancestor—not all functions need to create backtrace structures.*

The `return link value`, `return sp value` and `return fp value` are, respectively, the values to restore to pc, sp and fp at function exit.

The `save code pointer` must point 12 bytes beyond the start of the sequence of instructions that created the stack backtrace data structure.

20.3.3 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.

20.4 Control Arrival and Return

20.4.1 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 *20.3.1 The stack backtrace data structure* on page 20-4)
- sp points at or above the current stack limit. If the limit is explicit, sp will point at least 256 bytes above it (see *20.3 The Stack* on page 20-4)
- fp contains 0 or points to the most recently created stack backtrace structure (see *20.3.1 The stack backtrace data structure* on page 20-4)
- 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 *20.3 The Stack* on page 20-4)
- arguments are marshalled as described below

20.4.2 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
- a floating point value (of size 1, 2 or 3 words)

A callee may corrupt any of its arguments, howsoever passed.

At the instant control arrives at the target function, the argument list is allocated as follows:

- the first 4 argument words (or fewer if there are fewer than 4 argument words remaining in the argument list) are in machine registers a1-a4
- the remainder of the argument list (if any) are in memory, at the location addressed by sp and higher-addressed words thereafter

A floating-point value is treated as 1, 2 or 3 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.)

20.4.3 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 ➲*20.5 C Language Calling Conventions* on page 20-8).
- If the function returns a simple floating point value, the value is encoded in a1, a2 and a3.

20.5 C Language Calling Conventions

20.5.1 Argument representation

A floating point value occupies 1, 2, or 3 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 inter-working 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.

20.5.2 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.

20.5.3 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.

`struct {int a:8, b:8, c:8, d:8;}` and `union {int i; char *p;}` are both integer-like; `struct {char a; char b; char c; char d;}` is not.

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, ...);
```

20.6 Function Entry

20.6.1 Introduction

A complete discussion of function entry is complex; here we discuss a few of the most important issues and special cases.

The important issues for function entry are:

- creating the stack backtrace data structure (if needed)
- checking for stack overflow (if the stack limit is explicit)

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).*

20.6.2 Simple function entry - no stack backtrace structure

The entry sequence for functions that do not create a stack backtrace structure is simply:

```
PUSH {save-registers, lr} ; Save registers as needed.
```

Function exit is accomplished by the corresponding:

```
POP {save-registers, pc}
```

It may be necessary in some circumstances to save {a1-a4} before {v1-v4}. This may be necessary if the arguments may be addressed as a single array of arguments which is 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
```

20.6.3 Function entry - creating the stack backtrace structure

For non-variadic functions the stack backtrace structure can be created as follows:

```
SUB sp, sp, #16            ; create space for {fp, sp, lr, pc}
PUSH {save-registers}      ; save registers as necessary
MOV wr, pc
ADD wr, wr, #6             ; set up save code pointer from pc
STR wr, [sp, #pc-offset]   ; save into stack frame MOV wr, lr
STR wr, [sp, #lr-offset]   ; save lr into stack frame
ADD wr, sp, #<frame-size>  ; get original value of sp
STR wr, [sp, #<sp-offset>] ; and save into stack frame
MOV wr, fp
STR wr, [sp, #<fp-offset>] ; save fp into stack frame
ADD r7, sp, #<pc-offset>   ; set up new fp to point to save code
MOV fp, r7                 ; pointer
```

pc-offset, *lr-offset*, *sp-offset* and *fp-offset* are the offsets respectively of the saved pc, lr, sp and fp in the stack frame. These are defined as follows:

- *pc-offset* = No of saved regs * 4 + 12
- *lr-offset* = No of saved regs * 4 + 8
- *sp-offset* = No of saved regs * 4 + 4
- *fp-offset* = No of saved regs * 4 + 0

frame-size is the total size of the stack frame = No of saved regs * 4 + 16

There is no requirement to save argument registers a1-a4. These need only be saved by the called function where required.

Each of the registers v1-v4 only need be saved if it is going to be used by the called function. The minimum set of registers to be saved is {fp, old-sp, lr, pc}.

For variadic functions, a contiguous argument list can be made on the stack as follows:

```
PUSH {a1-a4}                          ; as necessary
SUB sp, sp, #16
...
ADD wr, sp, #frame-size+pushed-args   ; add in length of pushed
STR wr, [sp, #sp-offset]              ; args to get old sp
...
MOV fp, r7
```

It is not necessary to push arguments corresponding to fixed parameters.

20.6.4 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 on function entry (and perhaps at other times as well). There are two specific cases when this must be done:

- when the function uses 256 bytes or less of stack space
- when 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 only known at run-time. 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. IP is available as a work register. If the check fails, a standard run-time support function (`__rt_stkovf_split_small` or `__rt_stkovf_split_big`) is called. Each environment that supports explicit stack limit checking must provide these functions, which can do one of the following:

- terminate execution
- extend the existing stack, decrementing sl

20.6.5 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
```

20.6.6 Stack limit checking—large, fixed frames

For frames larger than 256 bytes, the limit check may be implemented as follows:

```
        LDR     wr, framesize
        ADD     ip, wr
        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 instants during their execution. If this is the case, subsequent stack adjustments require no limit check provided the initial stack check checks the maximum stack depth.*

20.7 Function Exit

Exit from functions is by means of the equivalent of a `MOV pc, lr` instruction, where lr has the same value as it had on entry to the function. `lr` does not need to be preserved.

Exit from functions which create a stack backtrace structure may be coded as follows:

```
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
```

21 File Formats

This section describes the file formats used by the ARM Software Toolkit.

File Formats

21.1 ARM Image Format

21.1.1 Properties 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 initialised static data

Two variants of AIF exist:

Executable AIF	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 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.

The remarks in the following subsection about executable AIF apply also to non-executable AIF, except that loader code must interpret the AIF header and perform any required decompression, relocation, and creation of zero-initialised data. Compression and relocation are, of course, optional: AIF is often used to describe very simple absolute images.

Executable AIF

It is assumed that on entry to a program in ARM Image Format (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 ARM Image Format 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, Acorn Absolute file types, etc.)
- read by the program loader from offset 0x28 in the file containing the AIF image
- given by some other means, eg. by instructing an operating system or debugger to load the image at a specified address in memory

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 discs). An AIF image is compressed by a separate utility which adds self-decompression code and data tables to it.

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 ➲*21.1.3 Zero-initialisation code* on page 21-7, allowing AIF to be easily tailored to new environments.

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 ➲*21.4 ARM Symbolic Debug Table Format* on page 21-32.

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. Thus, following relocation of a whole image, the debugging data area itself is position independent and may be copied or moved by the debugger.

21.1.2 The Layout of AIF

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.

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.

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-initialised 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 21-1: AIF header layout below*).

`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 shall not return... Non-executable AIF uses an offset, not BL	Note 2
`10:`	Program Exit Instr	...last ditch 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 (see note below).	Note 6
`28:`	Image base	Address the image (code) was linked at.	
`2C:`	Work space	Min 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 the image data was linked at.	
`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 21-1: 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 re-startable if, and only if, the program it contains is re-startable (an AIF image is *not* reentrant). If an AIF image is to be re-started then, 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 writeable, 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). (0 indicates an old-style 26-bit AIF header):

- the value 26, indicating the image was linked for a 26-bit ARM mode, and may not execute correctly in a 32-bit mode
- the value 32, indicating the image was linked for a 32-bit ARM mode, and may not execute correctly in a 26-bit mode

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 initialising data for the region follows the header.

9 *Debug Initialisation 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 customise 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.

21.1.3 Zero-initialisation code

The Zero-initialisation 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
```

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 fashion) and decides whether the image needs to be moved. If the image doesn't 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, being careful about the direction of copy, and jumping to the copied copy 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 the image was loaded at (the address of the AIF header) and the address the image was linked at (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 customise 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.

21.2 ARM Object Format

21.2.1 Introduction

This chapter defines a file format called *ARM Object Format* (AOF) which is used by language processors for ARM-based systems.

The ARM linker accepts input files in this 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 6, Linker*.

ARM Object Format directly supports the ARM Procedure Call standard (APCS), which is described in *Chapter 19, ARM Procedure Call Standard*.

21.2.2 About AOF

AOF is a simple object format, similar in complexity and expressive power to Unix's `a.out` format. It provides a generalised superset of `a.out`'s descriptive facilities. AOF was designed to be simple to generate and to process, rather than to be maximally expressive or maximally compact.

21.2.3 Terminology

In this chapter:

object file	refers to a file in ARM Object Format
linker	refers to the ARM linker
byte	8 bits; considered unsigned unless otherwise stated; usually used to store flag bits or characters
half word	16 bits, or 2 bytes; usually considered unsigned
word	32 bits, or 4 bytes; usually considered unsigned
string	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*

21.2.4 Byte sex or endianness

There are two sorts of AOF:

Little-endian AOF	The least significant byte of a word or half-word has the lowest address of any byte in the (half-)word. This *byte sex* is used by DEC, Intel and Acorn, amongst others.
Big-endian AOF	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 endian-ness 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). Thus the ARM linker will accept inputs of either sex and produce an output of the same sex, but will reject inputs of mixed endianness.

21.2.5 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 half-words 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.

21.2.6 The overall structure of an AOF file

An AOF file contains a number of separate but related pieces of data. To simplify access to these 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. The object file format defines five chunks:

- AOF header
- AOF areas
- producer's identification
- symbol table
- string table

These are described ➲*21.2.8 ARM object format* on page 21-13.

21.2.7 Chunk file format

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:

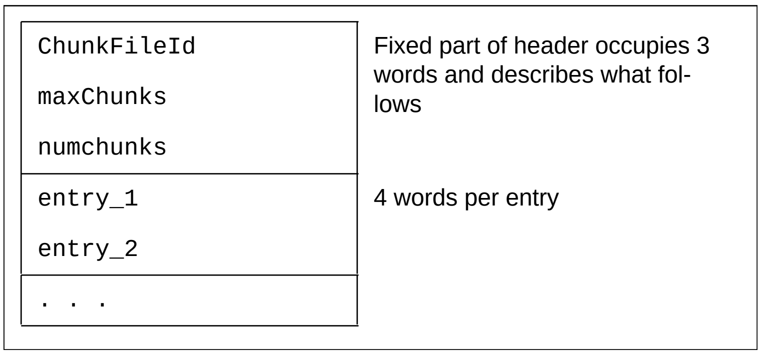


Figure 21-2: Chunk file layout

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, when read as a word, it appears to be 0xC5C6CBC3 then each word value must be byte- reversed before use).
maxChunks	defines the number of the entries in the header, fixed when the file is created. The *numChunks* field defines how many chunks are currently used in the file, which can vary from 0 to *maxChunks*.
numChunks	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);
fileOffset	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 8 characters are stored in ascending address order, as if they formed part of a NUL-terminated string, independently of endianness.

For AOF files, the first part of each chunk's name is "OBJ_"; the second components are defined in ➲*21.2.8 ARM object format* below.

21.2.8 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 *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 ➲*21.2.18 The string table chunk (OBJ_STRT)* on page 21-25). 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 (indeed, 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 5 chunks described here.

Note: The AOF header chunk should not be confused with the chunk file's header.

21.2.9 Format of the AOF header chunk

The AOF header consists of two parts, which appear contiguously in the header chunk.

First part	is of fixed size and describes the contents and nature of the object file.
Second part	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 following format:

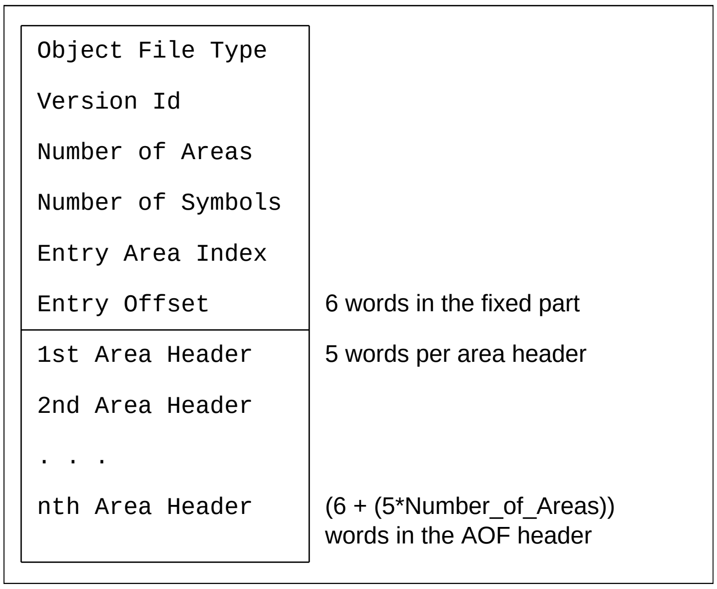


Figure 21-3: AOF header chunks

An `Object File Type` of 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 shall be identical to the endianness of the containing chunk file.

`Version Id` encodes the version of AOF to which the object file complies: version 1.50 is denoted by decimal 150; version 2.00 by 200; and this version 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), then `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.

21.2.10 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`	5 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 *21.2.18 The string table chunk (OBJ_STRT)* on page 21-25).
`Area Size`	gives the size of the area in bytes, which must be a multiple of 4. Unless the `Not Initialised bit` (bit 4) is set in the area attributes (see *21.2.11 Attributes + alignment* on page 21-16), there must be this number of bytes for this area in the OBJ_AREA chunk. If the `Not Initialised` bit is set, then there shall be no initialising bytes for this area in the OBJ_AREA chunk.
`Number of Relocations`	specifies the number of relocation directives which apply to this area, (equivalently: the number of relocation records following the area's contents in the OBJ_AREA chunk - see *21.2.13 Format of the areas chunk* on page 21-19).
`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.

21.2.11 Attributes + alignment

Each area has a set of attributes encoded in the most-significant 24 bits of the `Attributes + Alignment` word. The least-significant 8 bits of this word encode the alignment of the start of the area as a power of 2 and shall have a value between 2 and 32 (this value denotes that the area should start at an address divisible by $2^{\mathtt{alignment}}$).

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 ❯*6.4 Area Placement and Sorting Rules* on page 6-11.

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.
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 initialising data (see *Bit 12*, below). In effect, bit 11 implies bit 12. If both bits 10 and 11 are set, bit 11 is ignored.

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 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-initialised* attribute, specifying that the area has no initialising data in this object file, and that the area contents are missing from the OBJ_AREA chunk. Typically, this attribute is given to large uninitialised data areas. When an uninitialised area is included in an image, the linker either includes a read-write area of binary zeroes of appropriate size, or maps a read-write area of appropriate size that will be zeroed at image start-up time. This attribute is incompatible with the read-only attribute (see *Bit 13*, below). Whether or not a zero-initialised 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 initialised) 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 run-time, hardware permitting. Code areas and debugging tables should 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 run-time (usually, a debugger will extract 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-0 only if the area has the code attribute (bit 9 set). Bits 20-22 can be non-0 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 ARM Procedure Call Standard (APCS). For details, refer to ➲*19.3.1 32-bit PC vs 26-bit PC* on page 19-12. Such code may be incompatible with code which complies with a 26-bit variant of the APCS.
Bit 17	encodes the *reentrant* attribute, and denotes that code in this area complies with a reentrant variant of the ARM Procedure Call Standard.
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 ARM Procedure Call Standard without software stack-limit checking. Such code may be incompatible with code which 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 ➲*2.15 ARM/Thumb interworking* on page 2-57.

Bits 20-27 encode additional attributes of data areas, and must be non-0 only if the area does not have the code attribute (bit 9) unset. Bits 20-22 can be non-) 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 ➲*6.6.6 Based area relocation* on page 6-14).
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 initialised* attribute (bite 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 (refer to ➲*6.11 ARM Shared Library Format* on page 6-18).
Bits 22-23	are reserved and must be set to 0.
Bits 24-27	encode the base register used to address a *based* area. If the area does not have the *based* attribute then these bits shall be set to 0.
Bits 28-31	are reserved and must be set to 0.

21.2.12 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	Uninitialised(0-initialised)
13	0x00002000	Read only
14	0x00004000	Position independent
15	0x00008000	Debugging tables
		(Code areas only)
16	0x00010000	Complies with the 32-bit APCS
17	0x00020000	Reentrant code
18	0x00040000	Uses extended FP 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

Table 21-1: Area Attributes

Bit	Mask	Attribute Description
		(Data areas only)
20	0x00100000	Based area
21	0x00200000	Shared library stub data
24-27	0x0F000000	Base register for based area

Table 21-1: Area Attributes

21.2.13 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”. Pictorially, an area’s layout is:

```
Area 1

Area 1 Relocation

...

Area n

Area n Relocation
```

An area is simply a sequence of byte values. The endianness of the words and half-words within it must agree with that of the containing AOF file.

An area is followed by its associated table of relocation directives (if any). An area is either completely initialised by the values from the file or is initialised to zero, as specified by bit 12 of its area attributes.

Both the area contents and the table of relocation directives are aligned to 4-byte boundaries.

21.2.14 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						
1	II	B	A	R	FT	24-bit SID

`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 0-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 0-origin index in the array of areas, (or, equivalently, in the array of area headers).

The 2-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 half-word (2 bytes)
10	the field to be relocated is a word (4 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 suitably 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 *6.6 The Handling of Relocation Directives* on page 6-13.

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 way the relocation value is used to modify the subject field is determined by the `R` (PC-relative) bit, modified by the `B` (based) bit.

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
```

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 optimisation on reentrant code—for details, refer to ➲*6.6.4 Forcing use of an inter-link-unit entry point* on page 6-13.

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 reentrant code, the C compiler will place address constants in an adcon area based on register sb, and load 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 ➲*6.6.6 Based area relocation* on page 6-14.

Bits 29 and 30 of the relocation flags word must be 0; bit 31 must be 1.

21.2.15 The format of the symbol table chunk (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 only meaningful 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 field 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.

21.2.16 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. Specifically:

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 there were defined an identically-named symbol of global scope, having as its value 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 (ie. 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 (if 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, then 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 reentrant 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 ➲*6.6.4 Forcing use of an inter-link-unit entry point* on page 6-13.

Thumb:

Bit 12	is the Thumb attribute, which is set if the symbol is a Thumb symbol.

21.2.17 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 21-2: Symbol attributes

21.2.18 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 4 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 4, and no table has length less than 4.

The endianness of the length word must be identical to the endianness of the AOF and chunk files containing it.

21.2.19 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.

21.3 ARM Object Library Format

21.3.1 Introduction

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 ➲*21.2.7 Chunk file format* on page 21-11.

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 later in this document.

21.3.2 Terminology

The terms *byte, halfword, word*, and *string* are used to mean:

byte	8 bits, considered unsigned unless otherwise stated, usually used to store flag bits or characters
halfword	16 bits, or 2 bytes, usually considered unsigned
word	32 bits, or 4 bytes, usually considered unsigned
string	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.

21.3.3 Byte sex or endianness

There are two sorts of ALF: *little-endian* and *big-endian:*

Little-endian ALF	The least significant byte of a word or half-word has the lowest address of any byte in the (half-)word. This *byte sex* is used by DEC, Intel and Acorn, amongst others.
Big-endian ALF	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). Thus the ARM linker will accept inputs of either sex and produce an output of the same sex, but will reject inputs of mixed endianness.

21.3.4 Alignment

Strings and bytes may be aligned on any byte boundary.

ALF fields defined in this document do not use half-words, 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 half-words is defined by the use to which ALF is being put. For all current ARM-based systems, alignment is strict, as described immediately above.

21.3.5 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.

21.3.6 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:

`ChunkIndex`	
`EntryLength`	the size of this LIB_DIRY chunk (an integral number of words)
`DataLength`	the size of the Data (an integral number of words)
`Data`	

Figure 21-4: The LIB_DIRY chunk

where:

`ChunkIndex`	is a word containing the 0-origin index within the chunk file header of the corresponding LIB_DATA chunk. Conventionally, the first 3 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 0-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 2-word, word-aligned time stamp. The format of the time stamp is described in *21.3.7 Time stamps* on page 21-30. Its value is an encoded version of the last-modified time of the file from which the library member was created.

Earlier versions of ARM object library format

To 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 and not assumed to be sensible.
- 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.

21.3.7 Time stamps

A library time stamp is a pair of words encoding the following:

- a 6-byte count of centi-seconds since the start of the 20th century
- a 2-byte count of microseconds since the last centi-second (usually 0).

centi-seconds since 00:00:00 1st January 1900	first (most significant) word
u-seconds	second (least significant) word

The first word stores the most significant 4 bytes of the 6-byte count; the least significant 2 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 and is usually 0.

Time stamp words are stored in target system byte order: they must have the same endianness as the containing chunk file.

21.3.8 LIB_TIME

The LIB_TIME chunk contains a 2-word (8-byte) time stamp recording when the library was last modified.

21.3.9 LIB_VSRN

The version chunk contains a single word whose value is 1.

21.3.10 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.

21.3.11 Object code libraries

An object code library is a library file whose members are files in ARM Object Format (see *21.2 ARM Object Format* on page 21-10 for details).

An object code library contains two additional chunks: an external symbol table chunk named OFL_SYMT, and a time stamp chunk named OFL_TIME.

21.3.12 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 1 and 4 bytes of NUL padding, as follows:

`ChunkIndex`	
`EntryLength`	the size of this OFL_SYMT chunk (an integral number of words)
`DataLength`	the size of the External Symbol Name and Padding (an integral number of words)
`External Symbol Name`	
`Padding`	

OFL_SYMT entries do not contain time stamps.

21.3.13 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 ➲*21.3.7 Time stamps* on page 21-30).

21.4 ARM Symbolic Debug Table Format

21.4.1 Introduction

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.

For each separate compilation unit (called a *section*) the compiler produces debugging data, and a special *area* in the object code (see *21.2 ARM Object Format* on page 21-10 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 6, Linker*. For a description of the linker's principal output format see *21.1 ARM Image Format* on page 21-2.

The format of debugging data allows for a variable amount of detail. This potentially allows the user to trade off among memory used, disc 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 (see *21.4.7 Debugging data items in detail* on page 21-35). Low-level and high-level debugging are orthogonal facilities, though the debugger allows the user to move smoothly between levels if both sets of debugging data are present in an image.

21.4.2 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	(also called a *short*) 16 bits (2 bytes) Halfwords are unused, except in the long form of *LineInfo* items.

21.4.3 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; these items appear in an order corresponding to the order of definitions in the source. This means that any nested structure in the source program is preserved in the debugging data, and the debugger can use this structure to make deductions about the scope of various source-level objects. Of course, for procedure definitions, two debug items are needed:

procedure item	to mark the definition itself
endproc item	to mark 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.

21.4.4 Endianness and the encoding of debugging data

The ARM can be configured to use either a little-endian memory system (in which the least significant byte of each 4-byte word has the lowest address), or a big-endian memory system (in which the most significant byte of each 4-byte word has the lowest address).

In general, the code to be generated varies according to the bytesex (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 4-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).

21.4.5 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 to...; etc

For simple types the code is a positive integer as follows (all codes are decimal):

```
       void                0.
signed integers
       single byte         10
       half-word           11
       word                12
unsigned integers
       single byte         20
       half-word           21
       word                22
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 behaviour under debugger control is unlikely to be helpful to the user.

Fortran character types are supported by special kinds of debugging data item, the format of which is specific to each Fortran compiler.

21.4.6 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 (0-based) from the start of the line and the least-significant 22 bits give the line number.

21.4.7 Debugging data items in detail

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 recognise, it should use the length field to skip the item entirely. This discipline allows the debugging tables to be extended without invalidating existing debuggers.

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.

Offsets in file and addresses in memory

Where 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).

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 4 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

Lang byte

The language byte codes are defined in *Table 21-3: Language byte codes* below:

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 21-3: Language byte codes

All other codes are reserved to ARM.

flags byte

The `flags` byte uses the following mask values:

1	Debugging data contains line-number information
2	Debugging data contains information about top-level variables
3	Both of the above

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 file name (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 4 byte integer giving the number of symbols following.

Linker-generated debugging 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

Linker-generated debugging data: the section item is followed by `nsyms symbol` items, each consisting of 2 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 8 most significant bits, as shown in *Table 21-4: 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-initialised data
ASD_16BITSYM	0x10000000L	bit set if the symbol is a **Thumb** symbol

Table 21-4: 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 following 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 4. The end of the string table is padded with NULs to the next word boundary (so the length is a multiple of 4).

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 items contains the following word-sized fields:

`type`	the return type if this is a function, else 0 (see ➲*21.4.5 Representation of data types* on page 21-34)
`args`	the number of arguments
`sourcepos`	the source position of the procedure's start (see ➲*21.4.6 Representation of source file positions* on page 21-35)
`startaddr`	address of the first instruction of the procedure prologue The `startaddr` field addresses the start of the prologue. That is, the iinstruction at which 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, that is, the first address at which 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

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 ➲*21.4.6 Representation of source file positions* on page 21-35)
`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.

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.

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 ➲*21.4.5 Representation of data types* on page 21-34)
`sourcepos`	the source position of the variable (see ➲*21.4.6 Representation of source file positions* on page 21-35)
`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, which contains:

- an absolute address for static and external variables (relocated by the linker)
- a stack offset (an offset from the frame pointer) for automatic and var-type arguments
- an offset into the argument list for Fortran argument
- a register number for register variables (the 8 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).

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 ➲*21.4.5 Representation of data types* on page 21-34)
`name`	string

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 ➲*21.4.5 Representation of data types* on page 21-34)
`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.

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
`flags`	(see below)
`basetype`	a type word (described in ➲*21.4.5 Representation of data types* on page 21-34)
`lowerbound`	constant value or location of variable
`upperbound`	constant value or location of variable

If the size field is zero, debugger operations affecting the whole array, rather than individual elements of it, are forbidden.

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

Bounds

`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.

Note: *A variable item may be used to describe a location known to the compiler, which need not correspond to a source language variable.*

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:

`sizeandtype`	see below
`lb`	low bound of subrange
`hb`	high bound of subrange

The `sizeandtype` field 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 ➲*21.4.5 Representation of data types* on page 21-34) 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 might be held in 1 byte.)

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

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 *21.4.5 Representation of data types* on page 21-34)
`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)

Function declaration items

After its code and length word, a function declaration item contains the following fields:

`type`	a type word (see *21.4.5 Representation of data types* on page 21-34) 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 *21.4.5 Representation of data types* on page 21-34) 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.

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)

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:

`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 ofthe field in bits
`offset`	a byte containing the offset from bit 0 of the containing value of bit 0 of the field

followed by two zero bytes to pad to a word boundary.

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 that the body and argtable offsets are contained within the macro definition item, but the offset is, as normal, an offset within the containing section.

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

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)
`filename`	string (or "" if the name is not known)
`fragment data`	see below

If present, the date field contains the number of seconds since the beginning of 1970 (the Unix date origin).

Following the final file entry item, is a single 0 word marking 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 5 words, followed by a sequence of byte pairs and half word 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 a two or three (unsigned) half words, (each half word 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) half words.
`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 the line increment is non-zero (such an item may describe a block end or block start, for example).

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 described below.

Note: *There are two ways to describe 0 increments: 0 lines and 0 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 half word form is used, which in effect is a triple of half words 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.)

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 nofp APCS variant has been selected, or because they are sufficicently 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 e `-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 describedThese items are followed by a variable number of FPInfo fields., which take one of these forms:

(unsigned)	`byte codeinc`	size of code area described by this item
(signed)	`byte offsetinc`	change in the virtual FP offset for the next item

or, two zero bytes

(unsigned)	`short codeinc`
(signed)	`short offsetinc`

22 Remote Debugging

This chapter describes the Remote Debug Interface and the Remote Debug Protocol.

Remote Debugging

22.1 ARM Remote Debug Interface

22.1.1 Introduction

The Remote Debug Interface (RDI) is a concrete procedural interface between a debugger and a debuggee via a debug monitor or controlling debug agent. The interface can be 'pulled apart' to yield a pair of *stub* interfaces communicating via the Remote Debug Protocol (for details see ➲*22.2 ARM Remote Debug Protocol* on page 22-21).

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

Structure 1	This 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 operating system).
Structure 2	This would arise 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	These 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 instance via RS232 as used on the *PIE* card).

This chapter describes a C interface to the Remote Debug Protocol (RDP), designed to make using the RDP from a C program easier, and more efficient when the debugger and debug agent are linked as one program.

Every function returns an error status. Zero indicates no error, otherwise the value returned is the error number (see ➲*22.1.26 Error codes* on page 22-18).

It is the caller's responsibility to ensure that memory pointers do indeed point to valid memory locations in the debugger's address space.

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 so there can be good 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 have. These issues are highlighted in the following sections.

22.1.2 RDI functions

Purpose	**Function name**
Open and/or Initialise Debuggee	`RDI_open(type,config,hostos_if, dbg_state)`
Close and Finalise Debuggee	`RDI_close()`
Read Memory Address	`RDI_read(source,dest,nbytes)`
Write Memory Address	`RDI_write(source,dest,nbytes)`
Read CPU State	`RDI_CPUread(mode,mask,state)`
Write CPU State	`RDI_CPUwrite(mode,mask,state)`
Read Co-Processor State	`RDI_CPread(CPnum,mask,state)`
Write Co-Processor State	`RDI_CPwrite(CPnum,mask,state)`
Set Breakpoint	`RDI_setbreak(address,type, bound,point*)`
Clear Breakpoint	`RDI_clearbreak(point)`
Set Watchpoint	`RDI_setwatch(address,type, datatype,bound,point*)`
Clear Watchpoint	`RDI_clearwatch(point)`
Execute	`RDI_execute(point*)`
Multiple Step	`RDI_step(ninstr,point*)`
Break/watch inquiry	`RDI_pointinquiry(address*,type, datatype,bound*)`
Add Config Block	`RDI_addconfig(bytes)`
Load Config Block	`RDI_loadconfig(nbytes,data)`
Select Config Block	`RDI_selectconfig(aspect, name, type, version, req, versiongot)`
Get Driver Names	`RDI_drivernames()`
Get CPU Names	`RDI_cpunames()`
Get Error messages	`RDI_errmess(buf,len,errno)`

Load Debug Agent	`RDI_loadagent(dest,size,getf, getfarg)`
Miscellaneous Info	`RDI_info(type,arg1,arg2)`

22.1.3 Open and/or initialise debuggee

```
int RDI_open(unsigned type, struct Dbg_ConfigBlock const *config,
struct Dbg_HostasInterface const *i, struct Dbg_MCState *dbg_state)
```

`type`	distinguishes between types of initialisation:
	Bit 0 = 0 cold start (execute bootstrap, initialise MMU etc.).
	Bit 0 = 1 warm start (terminate execution, reset processor state etc.).
	Bit 1 = 1 reset the communication link.
`config`	This structure holds information such as the memory size, byte sex, serial port, processor, etc. See dbg_config.h for full details.
`i`	This structure 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`	This structure is internal to the debugger toolbox.

22.1.4 Close and finalise debuggee

`int RDI_close()`	terminate the current debugging session. Only a call to `RDI_open` may follow this call.

22.1.5 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.

22.1.6 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.

22.1.7 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.

`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, thus it is 8 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 (4 bytes per register). The requested registers are saved contiguously into this memory.

22.1.8 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.

22.1.9 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 is dependent 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 co-processor registers 0 to 7
  Bit 8 designates the floating point status register (FPSR)
  Bit 9 the floating point command register (FPCR)
- Co-processor 15 is a memory management unit (eg. ARM3's or ARM600's).

  Bits 0 to 7 of `mask` request transfer of MMU registers 0 to 7

22.1.10 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 is dependent 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 the floating point command register (FPCR)
- Co-processor 15 is a memory management unit (eg. ARM3's or ARM600's).

  Bits 0 to 7 of `mask` request transfer to MMU registers 0 to 7

22.1.11 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`. 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`. t*o* `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`.

Bit 4 of `type` if set indicates that the breakpoint is on a 16 bit (Thumb) instruction rather than a 32 bit (ARM) instruction.

In addition, bit 5 of `type` 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, no matter what the condition field of the breakpointed instruction.

Note that 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 ➲*22.1.25 Miscellaneous info* on page 22-11).

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` indicates that the call failed because the debugee currently has insufficient breakpoint resources available to honour this request.

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.

22.1.12 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`.

22.1.13 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. `type` defines the sort of watchpoint to set:

Type	**Halts on a data access to the address...**
`RDIPoint_EQ`	equal to `address`.
`RDIPoint_GT`	greater than `address`.
`RDIPoint_GE`	greater than or equal to `address`.
`RDIPoint_LT`	less than `address`.

`RDIPoint_LE`	less than or equal to `address`.
`RDIPoint_IN`	in the range from `address` to `bound`, inclusive.
`RDIPoint_OUT`	not in the range from `address` to `bound`, inclusive.
`RDIPoint_MASK`	halt execution if (`data-access-address` & `bound`) = `address`.

`datatype` defines the sort of data access to watch for:

`RDIWatch_ByteRead`	watch for byte reads
`RDIWatch_HalfRead`	watch for half-word reads
`RDIWatch_WordRead`	watch for word reads
`RDIWatch_ByteWrite`	watch for byte writes
`RDIWatch_HalfWrite`	watch for half-word writes
`RDIWatch_WordWrite`	watch for word writes

Values may be summed or ORed together in order to halt on any of a set of sorts of memory access. 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 ➲*22.1.25 Miscellaneous info* on page 22-11).

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 debugee currently has insufficient watchpoint resources to honour this request.

If a watchpoint is set on a location which already has one, the first watchpoint is removed before the new watchpoint is set.

22.1.14 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`.

22.1.15 Execute

```
int RDI_execute(PointHandle *point)
```

This function initiates execution in the debuggee, at the address currently loaded into the CPU Program Counter.

If a breakpoint is reached, or a watched address is accessed, or an exception occurs, or the user presses Escape, `RDI_execute` returns an error code (see ➲*22.1.26 Error codes* on page 22-18).

If a breakpoint or watchpoint caused the return, `*point` will be set to the handle identifying the break/watchpoint. At RDI specification level 0, a break/watch-point 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.

22.1.16 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 the user presses Escape, or the end of the program is reached before `instr` instructions have been executed, then `RDI_step` returns an error code indicating why execution was suspended (see ➲*22.1.26 Error codes* on page 22-18).

If a breakpoint or watchpoint caused the return, `*point` will be set to the handle identifying the break/watchpoint. At RDI specification level 0, a break/watch-point 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.

22.1.17 Break/watch-point inquiry

```
int RDI_pointinquiry(unsigned long *address, unsigned type,
      unsigned datatype, unsigned long *bound)
```

For inquiries about breakpoints, `datatype` must be 0. Otherwise, `type` and `datatype` are precisely as in corresponding calls to `setbreak` and `setwatch`.

`RDI_pointinquiry` returns information about what will happen if a corresponding call is made to `setbreak` or `setwatch`. (for range and comparison point types, the debuggee is permitted to do its best to honour the request and is not required to use precisely the address and bound requested).

If the break/watch type is supported then `address` and, if applicable, `bound` are updated to the values that will be used if a breakpoint or watchpoint is set. If it will be impossible to honour the corresponding break/watch-point request for lack of break/watch-point resources, the value `RDIError_NoMorePoints` is returned.

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 honoured, it is necessary to call `RDI_pointinquiry`.

22.1.18 Add config block

```
int RDI_addconfig(unsigned long bytes)
```

This function declares the size of a config block about to be loaded.

22.1.19 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 (eg ICEman).

22.1.20 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 reinitializes the Debug Agent to use the selected Config data.

`aspect`	one of RDI_ConfigCPU or RDI_ConfigSystem
`name`	the name of the configuration to be selected
`matchtype`	this specifies how exactly the version number must match that requested: RDI_MatchAny RDI_MatchExactly RDI_MatchNoEarlier
`versionreq`	the version number requested
`versionp`	the version actually selected

22.1.21 Get driver names

```
RDI_NameList const *RDI_drivernames(void)
```

where `RDI_NameList` is `typedef`'d to be a `struct` containing the number of names and an array of these names.

The returned names are used to recognise whether a particular driver has been selected on the command line.

22.1.22 Get CPU names

```
RDI_NameList const *RDI_cpunames(void)
```

This works in a similar way to `RDI_DriverNames`.

22.1.23 Get error messages

```
int RDI_ErrMess(char *buf,int buflen,int errno)
```

This requests that an error message (up to `buflen` characters) corresponding to `errno` is placed in buffer `buf`.

22.1.24 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`	This is the address in the Debug Agent's memory where the new version will be put.
`size`	This is the size of the new version, in bytes.
`getb`	This is 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.

22.1.25 Miscellaneous info

int RDI_info

```
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_Target

After a call of type `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 => inquires 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) required of the debugger by 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 identifiers the ARM processor or ARM emulator that the debuggee is running under.

Bits 5..10 allow a debugger to negotiate a suitable RDI specification level with a debuggee or to report gracefully that it is incompatible with the debuggee.

RDIInfo_Points

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 break/watch-points are supported
Bit 1:	range break/watch-points are supported
Bit 2:	watchpoints for byte reads are supported
Bit 3:	watchpoints for half-word reads are supported
Bit 4:	watchpoints for word reads are supported
Bit 5:	watchpoints for byte writes are supported
Bit 6:	watchpoints for half-word writes are supported
Bit 7:	watchpoints for word writes are supported
Bit 8:	mask break/watch-points 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, 8 apply only to breakpoints. Otherwise bits 0, 1, 8 apply to both breakpoints and watchpoints

All debuggees support breakpoints of sort `RDIPoint_EQ`, (break at specified address), so there is no bit denoting this in the value returned by `RDIInfo_Points`.

RDIInfo_Step

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.

RDIVector_Catch

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 1:	Undefined instruction
Bit 2:	Software interrupt (SWI)
Bit 3:	Prefetch abort
Bit 4:	Data abort
Bit 5:	Address exception
Bit 6:	Interrupt (IRQ)
Bit 7:	Fast interrupt (FIQ)
Bit 8:	Error

Bits which are 0 indicate that the corresponding exception vector should be taken by the debuggee.

RDISet_RDILevel

Set the RDI level to the value of `arg1`. The level set must lie between the limits returned by `RDIInfo_Target`.

RDISet_Thread

Set the thread context (for the interpretation of breakpoint and watchpoint requests) to the thread whose handle is given by `arg1`. The special value `RDINoHandle` resets the context to no thread, that is to the underlying hardware processor.

RDIInfo_MMU

This enquires about type and status of any MMU present. On return, arg1 contains a work which identifies the types of the MMU. `arg2` addresses a word describing the status of the MMU.

RDIInfo_DownLoad

This enquires whether configuration download and selection is available. The status returned is OK if these features are available.

RDIInfo_SemiHosting

This enquires whether `RDISemiHosting_* RDIInfo` calls are available. The status returned is OK if these features are available.

RDIInfo_CoPro

This enquires whether the `CoPro RDI Info` calls are available. The status returned is OK if these features are available.

RDIInfo_Icebreaker

This enquires whether the debuggee is controlled by ICEBreaker. The status returned is OK if the debuggee is controlled by ICEBreaker.

RDIMemory_Access

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

This call sets the characteristics for an area of memory. On entry `arg1` points to an array of <n> memory descriptions, `arg2` points to a word holding <n>. For full details of memory descriptions, see `RDI_MemDesc` in `dbg_stat.h`.

RDISet_CPUSpeed

This call sets the simulated CPU speed to be `arg1` (in nanoseconds).

RDIRead_Clock

This call reads the simulated CPU time. On return, `arg1` addresses the time in nanoseconds, and `arg2` the time in seconds.

RDIConfig_Count

This requests that the number of configuration blocks known to the debug agent should be returned to the word address by `arg1`. This option should only be used if `RDIInfo_Download` returned no errors.

RDIConfig_Ntl

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`. This option should only be used if `RDIInfo_Download` and `RDIConfig_Count` returned no errors.

RDIInfo_MemoryStats

This call enquires whether the last four calls are available (Memory_Access->Read_Clock). The status returned is OK is they are available.

RDIPointStatus_Watch

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

This is identical to `RDIPointStatus_Watch`, except that it is used for breakpoints.

RDISignal_Stop

This call requests that the debuggee stop execution.

RDISemiHosting_SetState

This should be used only if `RDIInfo_SemiHosting` did not return an error. `arg1` should point to 0 or 1 to disable/enable semihosting.

RDISemiHosting_GetState

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

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

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

This should be used only if `RDIInfo_ICEBreaker` did not return an error. A value indicating which ICEBreaker break points are locked is placed in the word pointed to by `arg1`.

RDIICEBreaker_SetLocks

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

This should be used only if `RDIInfo_Target` returned bit12 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

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 the Debug Comms Channel) to the host. <arg> is the value passed in `arg2` by `RDICommsChannel_ToHost`.

RDICommsChannel_FromHost

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 thehost 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

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 initialised.

RDIErrorP

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

Set the debuggee's command line arguments before commencing execution. `arg1` is a pointer to a NULL-terminated argument string, which must be no longer then 256 bytes including the NULL.

RDISet_RDILevel

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

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

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

THis function requests the description of the coprocessor the number of which is pointed to by `arg1`. `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

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

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

This should be used only if `RDIInfo_Target` returned bit 14 set in the value pointed to by `arg1` (to indicate that profiling is supported). This function specifies that profiling data should stop being collected.

RDIProfile_Start

This 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. This call starts profiling.

RDI_Profile_WriteMap

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

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

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.

22.1.26 Error codes

The symbolic values named here are defined in the file `rdi.h`. In each case, "`RDIError_`" must be prepended to the name shown below.

Error name	**Possible cause**
`NoError`	Everything worked
`Reset`	Debuggee reset
`UndefinedInstruction`	Tried to execute an undefined instruction
`SoftwareInterrupt`	A SWI happened
`PrefetchAbort`	Execution ran into unmapped memory
`DataAbort`	No memory at the specified address
`AddressException`	Accessed > 26 bit address in 26 bit mode
`IRQ`	An interrupt occurred
`FIQ`	A fast interrupt occurred
`Error`	A software error occurred
`BranchThrough0`	Branch through location 0
`NoMorePoints`	That's the last of the break/watchpoints
`CantSetPoint`	Break/watch-point resources exhausted
`BreakpointReached`	Returned by `RDI_execute` and `RDI_step`
`WatchpointAccessed`	Returned by `RDI_execute` and `RDI_step`
`ProgramFinishedInStep`	End of the program reached while stepping
`UserInterrupt`	User pressed Escape
`NoSuchPoint`	Tried to clear a break/watch point that did not exist
`CantLoadConfig`	configuration data could not be loaded
`BadConfigData`	configuration data was corrupted
`NoSuchConfig`	the requested configuration has not been loaded
`BufferFull`	buffer became full during operation
`OutOfStore`	Debug Agent ran out of memory
`NotInDownLoad`	illegal request made during DownLoad
`PointInUse`	ICEBreaker Breakpoint is already in use

`BadImageFormat`	Debug Agent could not make sense of AIF image supplied
`TargetRunning`	Target Processor did not stop (probably with BlackICE system)
`DeviceWouldNotOpen`	failed to open serial/parallel port
`NoSuchHandle`	no such memory description handle exists
`ConflictingPoint`	incompatible breakpoint already exists
`SoftInitialiseError`	recoverable error in RDI initialisation (You might need to use a different configuration.)

Information messages

The following are not really errors, but are just a means of passing information:

`LittleEndian`	The debuggee is little endian
`BigEndian`	The debuggee is big endian

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 can’t honour this RDP request
`UndefinedMessage`	Garbled RDP request
`IncompatibleRDILevel`	There is no common RDI level at which the debugger and debuggee can operate.

RDI errors

The following errors indicate misuse of the RDI or similar problem:

`NotInitialised`	`RDI_open` must be the first call
`UnableToInitialise`	The target world is broken
`WrongByteSex`	The debuggee can’t 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`	Don’t know what to do with this co-pro request

`BadCoProState`	Recognisably broken co-pro request.
`BadPointType`	Misuse of the RDI.
`UnimplementedType`	Misuse of the RDI.
`BadPointSize`	Misuse of the RDI.
`UnimplementedSize`	Halfwords not yet implemented

22.2 ARM Remote Debug Protocol

22.2.1 Introduction

The Remote Debug Protocol (RDP) is the communication protocol used between the ARM Symbolic Debugger and a remote debugee, via a debug monitor or controlling debug agent. Usually, the RDP is used via *stub* functions implementing the Remote Debug Interface (RDI): for details see ➲*22.1 ARM Remote Debug Interface* on page 22-2.

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

Structure 1	This arises in the variant of the symbolic debugger which is linked with ARM's standard ARM emulation environment (for the PC- and Sun-hosted cross-development variants of the debugger, and in the self-hosted, single address-space variant of *armsd* (for Acorn's RISC OS operating system). No direct use of the RDP is involved.
Structure 2	This would arise in an ARM-UNIX-hosted variant of the symbolic debugger, if the debugger and the ARM emulator (the *ARMulator*) were run in separate UNIX processes (perhaps on separate machines). In the second case, the *RDI* would consist of two stubs using UNIX's remote procedure calls to effect the inter-process message passing. Again, no direct use of the RDP is involved.
Structures 3 and 4	These 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 the debugger's host via a hardware debugging channel, (for instance via RS232 as used on the *PIE* card).

22.2.2 Terminology

PC and *pc* mean *program counter*. The address of the currently executing instruction in the debuggee.

An ARM processor can be configured to operate with either *little endian* memory (in which the least significant byte of a word has the lowest address of any byte in the word), or *big endian* memory (in which the most significant byte of a word has the lowest address of any byte in the word). The *endian-ness* of a memory system and processor configuration is also called its *byte sex*.

In the following sections, pseudo-C declarations are used to specify the content of messages and the types of arguments to message functions. In these declarations:

- *byte* means an 8 bit unsigned value
- *word* means an unsigned 4-byte value, transmitted least significant byte first (little endian)

The types *bytes* and *words* (plural) mean, respectively, a sequence of bytes and a sequence of words.

Values enclosed in { and } are present only in some contexts, as clarified by the explanatory text.

Each message of the RDP is encoded as a single function byte, followed immediately by its arguments, if any.

The return message acts as an acknowledgement as well as returning values. If the request is meaningful and successful, a zero status byte is returned, possibly preceded by requested data.

The reply to an unsatisfied request (failed request) is always padded to the same length that it would have had, had it been successful. Then follows a non-zero error code byte (see *22.2.7 Error codes* on page 22-42).

The RDP 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 so there can be good compatibility between debuggers and debug monitors released at different times. As a result, a debugger using the RDP must *negotiate* to establish its debuggee's capabilities and must not use capabilities its debuggee does not have. These issues are highlighted in the following sections.

22.2.3 Message summary

Debugger to debuggee message name	Hexadecimal Function Code
`Open and/or Initialise`	00
`Close and Finalise`	01
`Read Memory Address`	02
`Write Memory Address`	03
`Read CPU State`	04
`Write CPU State`	05
`Read Co-Processor State`	06
`Write Co-Processor State`	07
`Set Breakpoint`	0A
`Clear Breakpoint`	0B
`Set Watchpoint`	0C
`Clear Watchpoint`	0D
`Execute`	10
`Step`	11
`Info`	12
`OS Operation Reply`	13
`Add Configuration`	14
`Load Configuration`	15
`Select Configuration`	16
`Load Debug Agent`	17
`Interrupt Request`	18
`Comms Channel To Host (Reply)`	19
`Comms Channel From Host (Reply)`	1A
`Reset`	7F

Table 22-1: Debugger to debuggee messages

Debuggee to debugger message name	Hexadecimal Function Code
Stopped notification message	20
OS Operation Request	21
Comms Channel To Host	22
Comms Channel From Host	23
Fatal protocol error	5E
Return value/status message	5F
Reset	7F

Table 22-2: Debuggee to debugger messages

22.2.4 Debugger to debuggee messages

Open and/or initialise message (0)

```
Open(byte type, word memorysize {,byte speed})
return(byte status)
```

Upon receipt of this message a debuggee should prepare itself for an imminent debugging session, bootstrapping and/or initialising itself. This message will always be the first sent. If for some reason initialisation is impossible, a non zero status value should be returned. The `type` argument may be used to distinguish between sorts of initialisation:

Bit 0 = 0:	cold start (bootstrap, initialise MMU, etc.)
Bit 0 = 1:	warm start (terminate current execution, clear all breakpoints and watchpoints, etc.)
Bit 1 = 1:	reset the communication link
Bit 2 = 0:	debugger requires little endian debuggee
Bit 2 = 1:	debugger requires big endian debuggee
Bit 3 = 1:	debuggee should return its sex

The `memorysize` argument is used to specify the minimum number of bytes of memory that the debuggee's environment must have. A value of zero can be used if the debugger is not concerned with the memory size, (when the debuggee is running under an ARM emulator which allocates memory dynamically, as needed).

If bit 1 of the `type` argument is set, a single byte specifying the debug channel speed must follow the `memorysize` argument. A value of zero sets the default speed. Other values are target dependent. See ❯*Chapter 17, Demon* for possible return values.

The return value `RDIError_WrongByteSex` indicates that the debuggee has the opposite byte order to that requested in bit 2 of the type argument, and therefore the request has failed. If bit 3 of the `type` argument is set, the debuggee should ignore bit 2 and return a status of either `RDIError_LittleEndian` or `RDIError_BigEndian`.

Close and finalise message (1)

```
Close()
return(byte status)
```

Receipt of this message indicates the termination of the current debugging session. If for some reason the current debugging session cannot be terminated, a non-zero status value is returned. Only the Initialisation message may follow the Close message.

Read memory address message (2)

```
Read(word address, word nbytes)
return(bytes data, byte status {, word nbytes})
```

This message requests transfer of memory contents from the debuggee to the debugger. The transfer begins at `address`, and transfers `nbytes` of data in increasing address order.

On successful completion, the bytes requested are returned, followed by a zero status value.

On unsuccessful completion, the number of bytes requested are returned (some are garbage padding), followed by a non-zero error code byte, followed by the number of bytes successfully transferred. This number can be added to the base address to calculate the address where the transfer failed.

Write memory address message (3)

```
Write(word address, word nbytes, bytes data)
return(byte status {, word nbytes})
```

This message transfers data from the debugger to the debuggee's memory. The `address` argument specifies the location where the first byte of data is to be stored, and the `nbytes` argument gives the number of bytes to be transferred, followed by the bytes sequence to transfer.

A zero status value is returned on successful completion.

On failure, a non-zero error code byte is returned, followed by the number of bytes successfully transferred, just as with the `Read Memory Address` message.

Read CPU state message (4)

```
ReadCPU(byte mode, word mask)
return(words data, byte status)
```

This message is a request to read the values of registers in the CPU.

The `mode` argument defines the processor mode from which the transfer should be made. The mode number is the same as the mode number used by ARM6; a value of 255 indicates the current mode.

The `mask` argument indicates which registers should be transferred. Setting a bit to 1 will cause the designated register to be transferred.

Bit 0-14	request register R0-R14
Bit 15	requests the Program Counter (including the mode and flag bits in 26-bit modes)
Bit 16	requests transfer of the value of the Program Counter (without the mode and flag bits in a 26-bit mode)
Bit 17	requests the address of the currently executing instruction (often, 8 bytes less than the PC, because of instruction prefetching).
Bit 18	(in 32-bit modes) requests transfer of the CPSR; in 32-bit processor modes with an SPSR (non-user modes), bit 19 requests its transfer.
Bit 20	requests the transfer of the value of the flag and mode bits in a 26-bit mode (in the same bit positions as in register 15).

Upon successful completion, the values of the registers will be returned (the number depending on the number of bits set in the mask argument), followed by a zero status value. The lowest numbered register is transferred first.

On unsuccessful completion, the number of words specified in the mask is returned, followed by a non-zero error code byte.

Write CPU state message (5)

```
WriteCPU(byte mode, word mask, words data)
return(byte status)
```

This message is a request to set values of registers in the debuggee's CPU.

The `mode` argument defines the processor mode to which the transfer should be made.

The `mask` argument is as for the `ReadCPU` message, and is followed by the sequence of word values to be written to the registers specified in `mask`. The first value is written to the lowest-numbered register mentioned in `mask`.

The status value returned will be zero if the request was successful, otherwise, the error will be specified, (see *22.2.7 Error codes* on page 22-42).

Read co-processor state message (6)

```
ReadCoPro(byte CPnum, word mask)
return(words data, byte status)
```

This message is a request to read a co-processor's internal state. Its operation is similar to `ReadCPU`, except that register values are transferred from the co-processor numbered `CPnum`.

The registers to be transferred are specified by the `mask` argument.

The registers transferred, and their sizes, is co-processor-specific; 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 the transfer of floating point registers 0 to 7.

  Bit 8 requests the FPSR.

  Bit 9 requests the FPCR.
- Co-processor 15 is an MMU, for example ARM600's.

  Bits 0 to 7 of `mask` request the transfer of internal registers 0 to 7.

On successful completion, the values of the requested registers are returned followed by a zero status value. The lowest numbered register is transferred first.

On unsuccessful completion, the number of words implied by *mask* are transferred, followed by a non-zero error code byte.

For more details on the structure of a Coprocessor Description, including the format of coprocessor registers, see `dbg_cp.h`.

Write co-processor state message (7)

```
WriteCoPro(byte CPnum, word mask, words data)
return(byte status)
```

This message is a request to write a co-processor's internal state. This operation is similar to that of `WriteCPU`, except that register values are transferred to the co-processor numbered `CPnum`. The registers to be written are specified by the `mask` argument.

The registers transferred, and their sizes, depends on the co-processor; 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 the setting of floating point registers 0 to 7.

  Bit 8 requests a write to the FPSR.

  Bit 9 requests a write to the FPCR.
- Co-processor 15 is an MMU, for example ARM600's.

  Bits 0 to 7 of `mask` request the setting of internal registers 0 to 7.

The status value returned will be zero if the request was successful, otherwise the error is specified, (see ➲*22.2.7 Error codes* on page 22-42).

For more details on the structure of a Coprocessor Description, including the format of coprocessor registers, see `dbg_cp.h`.

Set breakpoint message (0x0A)

```
SetBreak(word address, byte type {, word bound})
return({word pointhandle} byte status)
  or  ({word address{word bound}} byte status)
```

This message requests the debuggee to set an execution breakpoint at `address`. The least significant 4 bits of `type` define the sort of breakpoint to set:

`0`	halt if the pc is equal to `address`.
`1`	halt if the pc is greater than `address`.
`2`	halt if the pc is greater than or equal to `address`.
`3`	halt if the pc is less than `address`.
`4`	halt if the pc is less than or equal to `address`.
`5`	halt if the pc is in the address range from `address` to `bound`, inclusive.
`6`	halt execution if the pc is not in the address range `address` to `bound`, inclusive.
`7`	halt execution if (pc & `bound`) = `address`.

At RDI/RDP specification levels 1 and above, bits 5, 6 and 7 of `type` have further significance.

Bit 4 of `type` set	If set, this indicates that the breakpoint is on a 16 bit (Thumb) instruction, rather than a 32 bit (ARM) instruction.
Bit 5 of `type` set	Requests that the breakpoint should be conditional (execution halts only if the breakpointed instruction is *executed*, not if it is conditionally skipped). If bit 5 is not set, execution halts whenever the breakpointed instruction is *reached* (whether executed or skipped).
Bit 6 of `type` set	Requests a dry run: the breakpoint is not set and the `address`, and if appropriate the `bound`, that would be used, are returned (for

	comparision and range breakpoints the `address` and `bound` used need not be exactly as requested). A zero status byte indicates that resources are currently available to set the breakpoint; `RDIError_NoMorePoints` indicates that the required breakpoint resources are not currently available.
Bit 7 of `type` set	Requests that a handle `pointhandle` should be returned for the breakpoint by which it will be identified subsequently. If bit 7 is set, a handle will be returned, whether the request succeeds or fails (but, obviously, it will only be meaningful if the request succeeds).

Note: Bits 6 and 7 must not be simultaneously set.

Upon successful completion a zero status byte is returned.

On unsuccessful completion, a non-zero error code byte is returned.

If the request is successful, but there are now no more breakpoint registers (of the requested type), then the value `RDIError_NoMorePoints` is returned.

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.

Clear breakpoint message (0x0B)

```
ClearBreak(word pointhandle)
return(byte status)
```

This message requests the clearing of the execution breakpoint identified by `pointhandle` which was set by an earlier `SetBreak` request (at level 0 of the RDI/RDP specification, `pointhandle` is the address at which the breakpoint was set).

Upon successful completion a zero status byte is returned.

On unsuccessful completion, a non-zero error code byte is returned.

Set watchpoint message (0x0C)

```
SetWatch(word address, byte type, byte datatype {, word bound})
return({word pointhandle} byte status)
  or  ({word address {,word bound}} byte status)
```

This message requests the debuggee to set a data access watchpoint at `address`. The least significant 4 bits `of type` define the sort of watchpoint to set:

`0`	halt on a data access to the address equal to `address`.
`1`	halt on a data access to an address greater than `address`.
`2`	halt on a data access to an address greater than or equal to `address`.
`3`	halt on a data access to an address less than `address`
`4`	halt on a data access to an address less than or equal to `address`.
`5`	halt on a data access to an address in the range from `address` to `bound`, inclusive.
`6`	halt on a data access to an address not in the range from `address` to `bound`, inclusive.

7 halt if (`data-acess-address` & `bound`) = `address`.

At RDI/RDP specification levels 1 and above, bits 6 and 7 of `type` have further significance.

Bit 6 of `type` set	Requests a dry run: the watchpoint is not set and the `address` and, if appropriate, the `bound`, that would be used, are returned (for range and comparison watchpoints, the `address` and `bound` used need not be exactly as requested). A zero status byte indicates that resources are currently available to set the watchpoint; `RDIError_NoMorePoints` indicates that the required watchpoint resources are not currently available.
Bit 7 of `type` set	Requests that a handle should be returned for the watchpoint by which it will be identified subsequently. If bit 7 is set, a handle will be returned, whether the request succeeds or fails (but, obviously, it will only be meaningful if the request succeeds).

Note: *Bits 6 and 7 must not be simultaneously set.*

The `datatype` argument defines the sort of data access to watch for:

`1`	watch for byte reads
`2`	watch for half-word reads
`4`	watch for word reads
`8`	watch for byte writes
`16`	watch for half-word writes
`32`	watch for word writes

Values may be summed or ORed together in order to halt on any of a set of sorts of memory access. For example:

8 + 16 + 32

to watch for any write access to the specified location(s).

Upon successful completion a zero status byte is returned.On unsuccessful completion, a non-zero error code byte is returned. If the request is successful, but there are now no more watchpoint registers (of the requested type), then the value `RDIError_NoMorePoints` is returned.

If a watchpoint is set on a location which already has a watchpoint, the first watchpoint will be removed before the new watchpoint is set.

Clear watchpoint message (0x0D)

```
ClearWatch(word pointhandle)
return(byte status)
```

This message requests the clearing of the data access watchpoint identified by `pointhandle`, which was set by an earlier `SetWatch` request (at level 0 of the RDI/RDP specification, `pointhandle` is the address at which the watchpoint was set).

Upon successful completion a zero status byte is returned. On unsuccessful completion, a non-zero error code byte is returned.

Execute message (0x10)

```
Execute(byte return)
return({word pointhandle} byte status)
```

This message requests that the debuggee commence execution at the address currently loaded into the CPU Program Counter.

If the least significant bit of `return` is 1, and commencing execution is viable, a return message is sent immediately and execution commences asynchronously. If the least significant bit of `return` is 0, then execution commences synchronously, and the return message is not sent until execution suspends:

- because the end of the program is reached
- because a break/watchpoint is reached
- because an exception occurs
- because the user interrupts the program

At RDI/RDP specification levels 1 and above, bit 7 of `return` has further significance: if it is set, an extra word will be returned which, if execution suspends because of a breakpoint or watchpoint, is the identifying handle of the breakpiont or watchpoint suspending execution.

On successful completion of the request, a zero status byte is returned. On completion of a synchronous request, a non-zero `status` byte may indicate the reason the debuggee suspended. Examples of possible return codes are:

2	Undefined Instruction
3	A SWI happened (only if watching for SWIs—see *Info message (0x12)* on page 22-31)
4	Prefetch Abort—instruction fetch from unmapped memory
5	Data Abort—no memory at the accessed address
6	Address Exception—26-bit mode access to address >= 2**26
7	IRQ
8	FIQ
9	Error
`10`	Branch through location 0
`143`	Breakpoint Reached It is the responsibility of RDP's caller to remove the breakpoint before continuing execution, or the debuggee will stop immediately at the same breakpoint.
`144`	Watchpoint Accessed It is not defined whether the PC addresses the accessing instruction or a

subsequent instruction, nor whether the accress has been performed.

`147`	User pressed Escape

Step message (0x11)

```
Step(byte return, word ninstr)
return({word pointhandle} byte status)
```

This message requests the debuggee to execute `ninstr` instructions, starting at the address currently loaded into the CPU Program Counter.

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 the least significant bit of `return` is 1, and starting execution is viable, then a return message is sent immediately and execution commences asynchronously.

If the least significant bit of `return` is 0, then execution start synchronously, and the return message is not sent until execution suspends, because:

- the requested number of instructions have been executed
- a break/watchpoint is reached
- an exception occurs
- the user interrupts the program

At RDI/RDP specification levels 1 and above, bit 7 of `return` has further significance: if it is set then an extra word will be returned which, if execution suspends because of a breakpoint or watchpoint, is the identifying handle of the breakpoint or watchpoint suspending execution.

On successful completion of the request, a zero status byte is returned.

On completion of a synchronous request, a non-zero `status` byte indicates the reason the debuggee suspended, exactly as for the Execute message, (see *Execute message (0x10)* on page 22-30).

Info message (0x12)

```
Info(word info {, argument})
return({words returninfo,} byte status)
```

This message requests the transfer of information between the debugger and the debuggee. The information transferred, and the direction of transfer depends on the value of `info`. In each case, a non-zero `status` byte indicates an unsuccessful request (see *22.2.7 Error codes* on page 22-42 for details).

info = 0: Returns information about the debuggee in the same way as:

```
return(word data, word model, byte status)
```

The value of `data` should be interpreted as follows:

Bit 16	1 => the debuggee has a communications channel
Bit 15	1 => the debuggee can cope with 16-bit (Thumb) code

Bit 14	1 => the Debug Agent can do profiling
Bit 13	1 => the Debug Agent supports RDP_Interrupt
Bit 12	1 => the Debug Agent supports enquiries about the download block size it supports
Bit 11	1 => the Debug Agent can be reloaded
Bits 8, 9, 10	the minimum RDI specification level (0-7) required of the debugger
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 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 of `model` is a unique identifier for the ARM processor or the emulator model that the debuggee is running under.

info = 1: Returns information about the debuggee's breakpointing and watchpointing capabilities, in the same way as:

```
return(word breakinfo, byte status)
```

The value of `breakinfo` should be interpreted as a set of bits, as follows:

Bit 0:	comparison breakpoints are supported
Bit 1:	range breakpoints are supported
Bit 2:	watchpoints for byte reads are supported
Bit 3:	watchpoints for half-word reads are supported
Bit 4:	watchpoints for word reads are supported
Bit 5:	watchpoints for byte writes are supported
Bit 6:	watchpoints for half-word writes are supported
Bit 7:	watchpoints for word writes are supported
Bit 8:	mask break/watch-points 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.

All debuggees must support breakpoints of sort RDIPoint_EQ (break at specified address).

Info = 2: Returns information about the debuggee's single-stepping capabilities, in the same way as:

```
return(word stepinfo, byte status)
```

The value of `stepinfo` 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.

info = 3: Returns information about the debuggee's memory management system (if any), in the same way as:

```
return(word meminfo, byte status)
```

The value of `meminfo` is a unique identifier for the type of memory manager used by the debuggee.

info = 4: enquires whether configuration download and selection are supported:

```
return (byte status)
```

A status return of 0 indicates that these facilities are supported.

info = 5: enquires whether info calls 0x181 to 0x184 (semi hosting Get/Set State/Vector) are supported:

```
return (byte status)
```

A status return of 0 indicates that these facilities are supported

info = 6: enquires whether info calls 0x400 and 0x401 (Describe Coprocessor and Request Coprocessor Description) are supported:

```
return (byte status)
```

A status return of 0 indicates that these facilities are supported.

info = 7: enquires whether the debuggee is controlled by ICEBreaker.

```
return (byte status)
```

A status return of 0 indicates that the debuggee is controlled by ICEBreaker.

info = 8: asks for the memory access statistics for the block of memory indicated by handle. For full details, see RDI_MemAccessStats in dbg_stat.h.

```
arguments(word handle)
```

```
return (word nreads, word nwrites, word sreads, word swrites, word
ns, word s, byte status)
```

info = 9: sets the characteristics for n regions of memory.

```
arguments(word n, {word handle, word start, word limit, byte width,
byte access, word Nread_ns, word Nwrite_ns, word Sread_ns, word
Swrite_ns}...)
```

```
return (byte status)
```

For further details, see dbg_stat.h.

info = 10: sets the simulated CPU speed in nanosecond

```
arguments(word speed)
```

```
return (byte status)
```

info = 12: reads the simulated CPU time

```
return (word ns, word s, byte status)
```

info = 13: enquires whether the Debug Agent supports info calls 8-12 (Memory Statistics)

```
return (byte status)
```

A status return of 0 indicates that these facilities are supported.

info = 14: enquires about the number of configuration blocks known to the Debug Agent:

```
return (byte_status, word_count)
```

The count is present only if the status indicates an error.

info = 0x80: returns the hardware resource number, and type of that resource when given a watchpoint handle. This can be used only if Info Call 1 (`Info_Points`) returns bit 12 (`RDIPointCapability_Status`) set.

```
arguments(word handle)
return (word hwresource, word type, byte status)
```

info = 0x81: is identical to info call 0x80, except that it works for a breakpoint handle.

info = 0x100: requests that the debuggee should immediately halt execution. If the debuggee is not executing or is running synchronously (see ➲*Execute message (0x10)* on page 22-30), a Return message and the status `RDIError_UserInterrupt` will be returned. If the debuggee is running asynchronously, then a `Stopped` message is returned, with status `RDIError_UserInterrupt`.

info = 0x180: informs the debuggee which hardware exceptions should be reported to the debugger; `argument` is a bit-mask of exceptions to be reported, as follows:

Bit 0:	Reset (branch through 0)
Bit 1:	Undefined Instruction
Bit 2:	Software Interrupt (SWI)
Bit 3:	Prefetch Abort
Bit 4:	Data Abort
Bit 5:	Address Exception
Bit 6:	Interrupt (IRQ)
Bit 7:	Fast Interrupt (FIQ)
Bit 8:	Error

A set bit in `argument` indicates that the exception should be reported to the debugger; a clear bit indicates the corresponding exception vector should be taken. When an exception is reported to the debugger, the state of the debuggee is rewound to the state pertaining just before executing the instruction which caused the exception.

info = 0x181: sets whether or not semihosting is enabled. It may be used only if Info call 5 (Info_SemiHosting) returned 0.

```
arguments (word semihostingstate)
```

```
result (byte status)
```

info = 0x182: reads whether or not semihosting is enabled. It may be used only if Info call 5 (Info_SemiHosting) returned 0.

```
result (word semihostingstate, byte status)
```

info = 0x183: sets the semihosting vector. It may be used only if Info call 5 (Info_SemiHosting) returned 0.

```
arguments (word semihostingvector)
result (byte status)
```

info = 0x184: reads the semihosting vector. It may be used only if Info call 5 (Info_SemiHosting) returned 0.

```
result (word semihostingvector, byte status)
```

info = 0x185: reads a bitmap which indicates which of ICEBreaker's breakpoints have been locked. This may be used only if Info Call 7 (Info_ICEBreaker) returned 0.

```
result (word lockedstate, byte status)
```

info = 0x186: writes a bitmap which indicates which of ICEBreaker's breakpoints are locked. This may be used only if Info Call 7 (Info_ICEBreaker) returned 0.

```
arguments(word lockedstate)
```

info = 0x187: requests the maximum length of data the Debug Agent can receive in one block. This may be used only if Info Call 0 (Info_Target) returned bit 12 set.

```
result (word maxloadsize, byte status)
```

info = 0x188: indicates whether data should be transferred from the debuggee to the debugger via the Debug Comms Channel. This may be used only if Info Call 0 (Info_Target) returned bit 16 set.

```
arguments(byte connect)
result (byte status)
```

info = 0x189: is the same as Info Call 0x188 except that it refers to data transfer from the debugger to the debuggee.

info = 0x200: Requests the debuggee to return the number of instructions and cycles executed since initialisation, in the same way as:

```
return (word ninstr, word S-cycles, word N-cycles, word I-cycles,
word C-cycles, word F-cycles, byte status)
```

info = 0x201: Requests the debuggee to return the error pointer associated with the last return to an Execute or Step request with status Error.Note that the error is returned as a word, rather than a byte.

info = 0x300: Requests that the debuggee's command line be set to `argument`. `argument` must be 0-terminated string of bytes and no longer than 256 bytes, including the terminating 0

info = 0x301: Requests that the RDI specification level be set to `argument`, a byte value lying between the limits returned by a call with `info` = 0. From receipt of an `open` request with bit 0 of `type` = 0 (a cold start open request) until receipt of this request, the debuggee operates with the RDI level set to its lower limit.

info = 0x302: Requests that the thread context (`SetBreak` and `SetWatch` messages) be set to `argument`, a 32 bit handle identifying a thread of execution. The distinguished handle `RDINoHandle` requests resetting the thread context to be the underlying hardware processor.

info = 0x400: Describes the registers of a coprocessor. `argument` has the form:

```
byte cpnum {byte rmin, byte rmax, byte nbytes, byte access,
            byte r0, byte r1, byte w0, byte w1}*
byte = 0xff
```

where:

`nbytes`	is the size in bytes of the register(s)
`access`	is a bitmask:

bit 0	register(s) readable with this bit set
bit 1	register(s) writeable with this bit set
bit 2	register(s) read or written via CPDT instructions (else CPRT) with this bit set. If bit2 is set, the registers provide bits as follows:

`r0`	bits 0 to 7
`r1`	bits16 to 23 of a CPRT instruction to read the register
`w0`	bits 0 to 7
`w1`	bits 16 to 23 of a CPRT instruction to write the register

Otherwise, `r0` provides bits 12 to 15 and `r1` bit 22 of CPDT instructions to read and write the register (and `w0` and `w1` are junk).

info = 0x401: Has `argument` as a byte coprocessor number. It requests that the debugee describe the registers of the coprocessor if it is known. The description is as by:

```
return( {byte rmin, byte rmax, byte nbytes, byte access}*, byte = 0xff)
```

where `rmin, rmax, nbytes` and `access` are as above.

info = 0x500: requests that profiling data should stop being collected. This should be used only if Info Call 0 (Info_Target) returned bit 14 set.

```
return (byte status)
```

info = 0x501: requests that profiling data should start being collected. This should be used only if Info Call 0 (Info_Target) returned bit 14 set.

```
arguments (word interval)
```

PC samples will be taken every `interval` microseconds.

info = 0x502: should be used only if Info Call 0 (Info_Target) returned bit 14 set.

```
arguments (word len, word size, word offset, words mapdata)
result (byte status)
```

This downloads a map array (described under RDI_Profile_WriteMap, *RDI_Profile_WriteMap* on page 22-17*)* which describes the PC ranges for profiling.

This is downloaded over the RDP in a series of messages:

`len`	the number of elements in the entire map array being downloaded
`size`	the number of map array elements being downloaded in this message
`offset`	the offset into the entire map array which this message starts from
`mapdata`	consists of `size` words of mapdata

info = 0x503: should be used only if Info Call 0 (Info_Target) returned bit 14 set.

```
arguments (word offset, word size)
result (words counts, byte status)
```

This uploads a a set of profile counts which correspond to the current profile map. See RDI_Profile_WriteMap, *RDI_Profile_WriteMap* on page 22-17*)* for more details.

This is uploaded over the RDP in a series of messages:

`offset`	the offset in the entire array of counts that this message starts from
`size`	the number of counts being uploaded in this message
`counts`	consists of `size` words of profiling counts data

info = 0x504: requests that profiling counts should all be reset to zero. This should be used only if Info Call 0 (Info_Target) returned bit 14 set.

```
result (byte status)
```

OS operation reply message (0x13)

```
OSOpReply(byte info, {data})
no reply
```

This message signals completion of the last requested OS Operation request.

`info` describes the type of the value returned by the operation:

0	return value
1	`data` comprises a single byte, to be placed in the debuggee's r0.
2	`data` comprises a word, to be placed in the debuggee's r0.

OS operations which return more complicated values must do so by using the appropriate combination of Write Memory and Write CPU state operations.

Add configuration message (0x14)

```
AddConfig(word nbytes)
return (byte status)
```

On receiving this message, the Debug Agent should prepare to receive a configuration data block of size n bytes. If the Debug Agent cannot accept a configuration data block of this size, it will return a non-zero status.

Load configuration message (0x15)

```
LoadConfigData(word nbytes, words data)
return (status)
```

On receiving this message, the Debug Agent should store away the configuration data ready for it to be selected. If there is an error during download, then a non-zero status will be returned. See the documentation of the Debug Agent concerned for more information about the content of the configuration data.

Select configuration message (0x16)

```
SelectConfig(byte aspect, byte namelen, byte matchtype, word vsn_req,
bytes name)
return (word vsn_sel, byte status)
```

On receiving this message, the Debug Agent should select one of the sets of configuration data blocks and reinitialise the Debug Agent to use that configuration:

`aspect`	one of RDI_ConfigCPU or RDI_ConfigSystem
`namelen`	the number of bytes in name
`name`	`namelen` bytes which comprise the name of the configuration
`vsn_req`	the requested version of the named configuration
`matchtype`	specifies how the selected version must match that specified, and should be one of RDI_MatchAny, RDIMatchExactly, or RDI_MatchNoEarlier

The Debug Agent returns the version number of the configuration selected and a byte of status.

Load debug agent (0x17)

```
LoadAgent(word loadaddress, word size)
return (byte status)
```

On receiving this message, the Debug Agent should prepare to receive configuration data which it should interpret as a new version of the Debug Agent code. The new Debug Agent code will be `size` bytes long, and should be loaded at `loadaddress`. The data will be downloaded using LoadConfigurationData (0x15).

Interrupt execution (0x18)

```
Interrupt()
no return
```

On receiving this message, the Debug Agent should attempt to stop execution of the debuggee. No return value should be sent.

CCToHostReply (0x19)

```
CCToHostReply()
return (byte status)
```

On receiving this message, the Debug Agent knows whether or not the host successfully received the data it was sent to CCToHost (0x22).

CCFromHostReply (0x20)

```
CCFromHostReply(byte valid, word data)
return (byte status)
```

This message returns data to be sent to the debuggee using the Debug Comms channel. If `valid` is 0, there is no data, otherwise `data` is the word to be transferred.

Reset message (0x7F)

```
RequestReset()
no reply
```

Requests that the debuggee reset itself.

22.2.5 Debuggee to debugger messages

The debugger does not acknowledge debuggee-to-debugger messages, as there is no point in trying to handle errors in the debuggee.

All responses to debugger-to-debuggee requests are of the `Return` message type, described in ➲*Return message (0x5F)* on page 22-41.

Stopped message (0x20)

```
Stopped({word pointhandle} byte status)
```

This message is sent to a debugger by an asynchronously executing debuggee, to indicate that execution of the debuggee has suspended. Execution of the debuggee was previously started by an `Execute(X)` or `Step(X, n)` message bith bit 1 of `X` = 1.

The `status` value indicates the type of suspension; for details, see ➲*Execute message (0x10)* on page 22-30.

At RDI specification levels 1 and above, if bit 7 of `X` was 1, a word `pointhandle` is returned. If execution suspended because a breakpoint was reached or a watchpoint was accessed, then the value of `pointhandle` identifies the -point concerned.

OS operation request message (0x21)

```
OSOp(word op, byte argdesc, {args})
```

This message is sent by the debuggee to request execution of a call to the host operating system.

`op`	identifies which call
`argdesc`	allows description of up to 4 arguments to the call: if there are more, or their format is more complicated than can be described by `argdesc`, their values must be acquired by the appropriate combination of Read Memory and Read CPU State operations. `argdesc` is a sequence of two-bit fields starting from the least significant end of the word, each of which describes the corresponding argument, as follows:

`0`	no such argument
`1`	a single byte
`2`	a word
`3`	a string

The format of a string value is determined by its first byte:

`0-32`	the string is of this length, and its component bytes follow, (the terminating 0 byte is omitted).
`33-254`	the string is of this length. A word containing the address of the string follows. The read memory message can be used to access the string.
`255`	a word follows containing the string's length, then a word containing its address. The read memory message can be used to access the string.

Comms channel to host message (0x22)

```
CCToHost (word data)
return (byte status)
```

This message sends a word of data which has been transferred from the debuggee via the Debugs Comms Channel up to the host.

Comms channel from host message (0x23)

```
CCFromHost ()
return (byte valid, word data, byte status)
```

This message requests a word of data from the host to be sent to the debuggee via the Debug Comms Channel. If `valid` is 0, then the host has no data to transfer. Otherwise, `data` is valid.

Fatal message (0x5E)

```
Fatal(byte error)
```

The `Fatal` message indicates that the debuggee could not make sense of the last message sent.

Return message (0x5F)

`Return(..., byte status)`

The `Return` message is used to acknowledge a recognised request message.

A `status` value is always returned, indicating that the syntax of the original request was understood, and whether or not it was satisfied.

The arguments to `Return` depend on the message being acknowledged, and are described in ➲*22.2.4 Debugger to debuggee messages* on page 22-24.

Reset message (0x7F)

This message indicates that the debuggee has reset itself, either because of a hardware reset, or in response to a reset request.

22.2.6 Notes

Address spaces

In debuggee environments that support different address spaces in different processor modes, the address space corresponding to the processor mode at the time the message is sent is used by all memory access, breakpoint and watchpoint instructions.

Minimum support

There is a minimum subset of requests that all debuggees must support. The info message is used to enquire whether a debuggee can support operations outside the minimum subset, consisting of:

Message	Function code
Open and/or Initialise	00
Close and Finalise	01
Read Memory Address	02
Write Memory Address	03
Read CPU State	04
Write CPU State	05
Set Breakpoint	0A (with first argument = 0)
Clear Breakpoint	0B
Execute	10
Info	20
Reset	7F

22.2.7 Error codes

Debuggee status

The following error code values indicate debuggee status:

Code	Error Name	Possible cause
0	`No error`	Everything worked
1	`Reset`	Debuggee reset
2	`Undefined instruction`	Tried to execute the undefined instr
3	`Software interrupt`	A SWI happened (when tracing SWIs)
4	`Prefetch abort`	Execution ran into unmapped memory
5	`Data abort`	No memory at the specified address
6	`Address exception`	Accessed > 26 bit address in 26 bit mode
7	`IRQ`	An interrupt occurred
8	`FIQ`	A fast interrupt occurred
9	`Error`	An error occurred
10	`BranchThrough0`	Branch through location 0
142	`No more points`	That’s the last of the break/watchpoints
143	`Breakpoint reached`	What Execute and Step can return...
144	`Watchpoint accessed`	... or this
146	`Program finished`	End of the program reached while stepping
147	`User interrupt`	User pressed Escape
148	`CantSetPoint`	Break/watch-point resources exhausted
150	`CantLoadConfig`	Configuration Data could not be loaded
151	`BadConfigData`	Configuration data was corrupt
152	`NoSuchConfig`	The requested configuration has not been loaded.
153	`BufferFull`	Buffer was filled during the operation.

Table 22-3: Debuggee status

Code	Error Name	Possible cause
154	`OutOfStore`	The Debug Agent ran out of memory.
155	`NotInDownLoad`	Illegal request made during download.
156	`PointInUse`	ICEBreaker breakpoint is already being used.
157	`BadImageFormat`	Debug Agent could not make sense of AIF image supplied.
158	`TargetRunning`	Target processor could not be halted (probably with EmbeddedICE system).
159	`DeviceWouldNotOpen`	Failed to open serial or parallel port.
160	`NoSuchHandle`	No such memory description handle exists.
161	`ConflictingPoint`	Incompatible breakpoint already exists.

Table 22-3: Debuggee status (Continued)

Internal fault or limitation

The following errors indicate an internal fault or limitation:

Code	Error Name	Possible cause
253	`InsufficientPrivilege`	Supervisor state was not accessible to this debug monitor
254	`Unimplemented message`	Debuggee can't honour this RDP request
255	`Undefined message`	Garbled RDP request

Table 22-4: Internal faults

Information messages

The following errors are not really errors, but are just a means of passing information:

Code	Error Name	Possible cause
240	`LittleEndian`	The debuggee is little endian
241	`BigEndian`	The debuggee is big endian
242	`SoftInitialiseError`	A recoverable error occurred during initialisation. Perhaps different configuration data is required.

Table 22-5: Information

Misuse of the RDI

The following errors indicate misuse of the RDI or similar problem:

Code	Error Name	Possible cause
128	`Not initialised`	Open must be the first call
129	`Unable to initialise`	The target world is broken
130	`WrongByteSex`	The debuggee can’t operate with the requested byte sex
131	`Unable to terminate`	Target world was smashed by the debuggee
132	`Bad instruction`	It is illegal to execute this instruction
133	`Illegal instruction`	The effect of executing this is undefined
134	`Bad CPU state`	Tried to set the SPSR of user mode
135	`Unknown co-processor`	This co-processor is not connected
136	`Unknown co-proc state`	Don’t know how to handle this request
137	`Bad co-proc state`	Recognisably broken co-proc request

138
139
140
	`Bad point type`
	`Unimplemented type`
	`Bad point size`
		Misuse of the RDI

141	`Unimplemented size`	Half words not yet implemented
145	`No such point`	Tried to clear an unset break/watchpoint

Table 22-6: RDI error codes

Index

Index

Index

Index

Index