ARM Developer Suite

Version 1.0.1

Tools Guide

ARM DUI 0067B

Release Information

The following changes have been made to this book.

Change History

Date	Issue	Change
October 1999	A	Release 1.0
March 2000	B	Release 1.0.1

Proprietary Notice

Contents
Tools Guide

Preface

This preface introduces the *ARM Developer Suite* (ADS) tools and reference documentation. It contains the following sections:

- *About this book* on page Preface-ii
- *Feedback* on page Preface-vii

-

About this book

This book provides reference information for ADS. It describes the command-line options to the assembler, linker, compilers and other ARM tools in ADS. The book also gives reference material on the ARM implementation of the C and C++ compilers and the C libraries.

Intended audience

This book is written for all developers who are producing applications using ADS. It assumes that you are an experienced software developer and that you are familiar with the ARM development tools as described in ADS *Getting Started*.

Using this book

This book is organized into the following chapters:

Chapter 1 *Introduction*

Read this chapter for an introduction to ADS.

Chapter 2 *C and C++ Compilers*

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

Chapter 3 *ARM Compiler Reference*

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

Chapter 4 *The C and C++ Libraries*

Read this chapter for a description of the ARM C and C++ libraries and instructions on re-implementing individual library functions.

Chapter 5 *Assembler*

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

Chapter 6 *The ARM Linker*

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

Chapter 7 *Toolkit Utilities*

Read this chapter for a description of the utility programs provided with ADS, including fromELF, the ARM profiler, the ARM librarian, and the ARM file downloaders.

Chapter 8 *Floating-point Support*

Read this chapter for a description of floating-point support in ADS.

Typographical conventions

The following typographical conventions are used in this book:

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

Further reading

This section lists publications from both ARM Limited and third parties that provide additional information on developing code for the ARM family of processors.

ARM periodically provides updates and corrections to its documentation. See `http://www.arm.com` for current errata sheets and addenda.

See also the ARM Frequently Asked Questions list at:
`http://www.arm.com/DevSupp/Sales+Support/faq.html`

ARM publications

This book contains reference information that is specific to development tools supplied with ADS. Other publications included in the suite are:

-

- *Getting Started* (ARM DUI 0064A)
- *ADS Developer Guide* (ARM DUI 0056A)
- *ADS Debuggers Guide* (ARM DUI 0066A)
- *ADS Debug Target Guide* (ARM DUI 0058A)
- *CodeWarrior IDE Guide* (ARM DUI 0065A).

The following additional documentation is provided with the ARM Developer Suite:

- *ARM Architecture Reference Manual* (ARM DUI 0100). This is supplied in Dynatext format, and in PDF format in `install_directory\PDF\ARM-DDI0100B_armarm.pdf`.
- *ARM Applications Library Programmer's Guide* (ARM DUI 0081). This is supplied in Dynatext format, and in PDF format on the CD.
- *ARM ELF specification* (SWS ESPC 0003). This is supplied in PDF format in `install_directory\PDF\specs\ARM ELFA08.pdf`.
- *TIS DWARF 2 specification*. This is supplied in PDF format in `install_directory\PDF\specs\TIS-DWARF2.pdf`.
- *Angel Debug Protocol*. This is supplied in PDF format in `install_directory\PDF\specs\ADP ARM-DUI0052C.pdf`
- *Angel Debug Protocol Messages*. This is supplied in PDF format in `install_directory\PDF\specs\ADP ARM-DUI0053D.pdf`

In addition, refer to the following documentation for specific information relating to ARM products:

- *ARM Reference Peripheral Specification* (ARM DDI 0062)
- the ARM datasheet or technical reference manual for your hardware device.

Other publications

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

The following book gives general information about the ARM architecture:

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

The following books describe the C++ language:

- *ISO/IEC 14882:1998(E), C++ Standard*. Available from the national standards body.

The following books provide general C++ programming information:

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

  This is a reference guide to C++.

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

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

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

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

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

  The sequel to *Effective C++*.

The following books provide general C programming information:

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

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

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

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

-

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

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

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

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

Feedback

ARM Limited welcomes feedback on both ADS and the documentation.

Feedback on the ARM Developer Suite

If you have any problems with ADS, please contact your supplier. To help them provide a rapid and useful response, please give:

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

Feedback on this book

If you have any problems with this book, please send email to `errata@arm.com` giving:

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

General suggestions for additions and improvements are also welcome.

Chapter 1
Introduction

This chapter introduces the *ARM Developer Suite* (ADS). It contains the following sections:

- *About the ARM Developer Suite* on page 1-2
- *Supported platforms* on page 1-5.

1.1 About the ARM Developer Suite

ADS consists of a suite of applications, together with supporting documentation and examples, that enable you to write and debug applications for the ARM family of RISC processors.

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

1.1.1 Components of the toolkit

ADS toolkit consists of the following major components:

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

These are described in more detail below.

Command-line development tools

The following command-line development tools are provided:

armcc	The ARM C compiler. The compiler is tested against the Plum Hall C Validation Suite for ANSI conformance. It compiles ANSI source into 32-bit ARM code in ELF object format.
armcpp	This is the ARM C++ compiler. It compiles ANSI C++ or EC++ source into 32-bit ARM code in ELF object format.
tcc	The Thumb C compiler. The compiler is tested against the Plum Hall C Validation Suite for ANSI conformance. It compiles ANSI source into 16-bit Thumb code in ELF object format.
tcpp	This is the Thumb C++ compiler. It compiles ANSI C++ or EC++ source into 16-bit Thumb code in ELF object format.
armasm	The ARM and Thumb assembler. This assembles both ARM assembly language and Thumb assembly language source into ELF object format.
armlink	The ARM and Thumb linker. This combines the contents of one or more object files with selected parts of one or more object libraries to produce an executable program. The ARM linker creates ELF executable images.

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

Rogue Wave C++ library

The Rogue Wave library provides standard C++ functions and objects such as `cout()`. For more information on Rogue Wave, see the online documentation.

support libraries

The ARM C libraries provide additional components to enable support for C++ and to compile code for different architectures and processors.

See Chapter 2 *C and C++ Compilers*, Chapter 5 *Assembler*, and Chapter 6 *The ARM Linker* for more information on the command-line development tools. See Chapter 4 *The C and C++ Libraries* for more information on the libraries.

GUI development tools

The following GUI development tools are provided:

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

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

ADU The old ARM Debugger for UNIX. This provides a full GUI environment for debugging your C, C++, and assembly language source.

CodeWarrior IDE

The project manager for Windows. This is a graphical user interface tool that automates the routine operations of managing source files and building your software development projects. The CodeWarrior IDE helps you to construct the environment, and specify the procedures needed to build your software.

See the *ADS Debuggers Guide* and the *CodeWarrior IDE Guide* for more information on the development tools.

Utilities

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

fromELF	The ARM image conversion utility. This accepts ELF format input files and converts them to a variety of output formats, including AIF, plain binary, *Extended Intellec Hex* (IHF) format, Motorola 32-bit S-record format, and Intel Hex 32 format. The utility can also produce textual information about the input file, code disassembly for example.
armprof	The ARM profiler displays an execution profile of a program from a profile data file generated by an ARM debugger.
armar	The ARM librarian enables sets of ELF files to be collected together and maintained in libraries. You can pass such a library to the linker in place of several ELF object files.
Flash downloader	The Flash downloader enables you to download binary images to the flash memory of supported ARM development and evaluation boards.

See Chapter 7 *Toolkit Utilities* for more information on the utilities.

Supporting software

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

ARMulator	The ARM core emulator. This provides instruction-accurate emulation of ARM processors, and enables ARM and Thumb executable programs to be run on non-native hardware. The ARMulator is integrated with the ARM debuggers.
Angel	The ARM debug monitor. Angel runs on target development hardware and enables you to develop and debug applications running on ARM-based hardware. Angel can debug applications running in either ARM state or Thumb state.

See the *ADS Debuggers Guide* and the *ADS Debug Target Guide* for more information on the supporting software.

The ARM debuggers also support the hardware emulation tools Multi-ICE and EmbeddedICE. These products are available separately.

1.2 Supported platforms

This release of the ADS is supported on the following platforms:

- Sun workstations running Solaris 2.5.1 or 2.6
- Hewlett Packard workstations running HP-UX 10.20
- IBM-compatible PCs running Windows 95, Windows 98, or Windows NT 4.

The CodeWarrior IDE is supported on IBM-compatible PCs running Windows 95, Windows 98, and Windows NT 4.

Chapter 2
C and C++ Compilers

This chapter provides details of the command-line options to the ARM and Thumb, C and C++ compilers. Although equivalent functionality is available from Windows dialogs, this chapter assumes you are familiar with command-line software development tools such as those provided with ADS. For an introduction to command-line development, see the *ADS Developer Guide*.

This chapter contains the following sections:

2.1 About the C and C++ compilers

Wherever possible, the compilers adopt widely used command-line options familiar both to users of UNIX and to users of Windows/MS-DOS.

The ARM C and C++ compilers compile ANSI C.

The ARM C++ compilers expect C++ that conforms to the ISO/IEC 14822 :1998 International Standard for C++. See *Standard C++ implementation definition* on page 3-30 for a detailed description of ARM C++ language support.

The ARM C++ compilers can also compile the subset of standard C++ known as *Embedded C++* (EC++). EC++ is a subset of standard C++ that provides efficient code for use in embedded systems. The EC++ amendment to the ISO standard is still evolving. The proposed definition can be found on the web at `http://www.caravan.net/ec2plus`.

2.1.1 Compiler variants

All ARM C and C++ compilers accept the same basic command-line options. Unless stated otherwise, the text in this chapter applies to all compiler types. Where a specific compiler has added features or restrictions, this is noted in the text. Where an option applies only to C++, this is also noted in the text.

There are four compiler variants as shown in Table 2-1:

Table 2-1 Compiler variants

Compiler name	Compiler variant	Source language	Compiler output
armcc	C	C	32-bit ARM code
tcc	C	C	16-bit Thumb code
armcpp	C++	C or C++	32-bit ARM code
tcpp	C++	C or C++	16-bit Thumb code

Note

Throughout this chapter, the phrase *the ARM compilers* refers to armcc, armcpp, tcc, and tcpp.

2.1.2 Source language modes

The ARM compilers have three distinct source language modes that can be used to compile several varieties of C and C++ source code:

ANSI C In ANSI C mode, the ARM compilers have passed release 7.00 of the *Plum Hall C Validation Suite* (CVS). This suite has been adopted by the British Standards Institute for C compiler validation in Europe. The compiler option `-strict` was used when running the tests.

EC++ This mode applies only to the ARM C++ compilers. The ARM C++ compilers compile the Embedded C++ subset of the ISO/IEC Standard C++.

C++ This mode applies only to the ARM C++ compilers. The ARM C++ compilers compile ISO/IEC standard C++. The compilers are tested against *Suite++, The Plum Hall Validation Suite for C++, version 5.00*. This is the default language mode for the ARM C++ compilers. The option `-strict` was used when running the tests.

For more information on how to use compiler options to set the source mode for the compiler, see *Setting the source language* on page 2-12.

2.1.3 Library support

ADS provides both ANSI C libraries in prebuilt binary form and Rogue Wave C++ libraries in prebuilt binary form. See Chapter 4 *The C and C++ Libraries* for detailed information about the libraries.

You can create you own definition of target-dependent functions in order to customize the C libraries. Most retargeting is done automatically by setting the compiler options for processor architecture and family.

-

2.2 File usage

This section describes naming conventions and included files.

2.2.1 Naming conventions

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

The ARM compilers use or generate files with the following file suffixes:

`filename``.c`	ARM C compilers recognize the `.c` suffix as source files. ARM C++ compilers recognize `.c`, `.cpp`, `.cp`, `.c++` and `.cc` suffixes as source files.
`filename``.h`	header file (a convention only, this suffix has no special significance for the compiler).
`filename``.o`	ARM object file.
`filename``.s`	ARM or Thumb assembly language file. (This can be placed in the input file list or, with the `-S` option, produced as an output file.)
`filename``.lst`	error and warning list file (the default output extension for `-list` option).

Portability

The ARM compilers support multiple file-naming conventions on all supported hosts. To ensure portability between hosts, use the following guidelines:

- ensure that filenames do not contain spaces. If you have to use pathnames or filenames containing spaces, enclose the path and filename in quotes.
- make embedded pathnames relative rather than absolute.

In each host environment, the compilers support:

- native filenames
- pseudo UNIX filenames in the format:

  `host-volume-name:/rest-of-unix-file-name`
- UNIX filenames using `/` as a path separator.

Filenames are parsed as follows:

- a name starting with `host-volume-name:/` is a pseudo UNIX filename
- a name that does not start with `host-volume-name:/` and contains `/` is a UNIX filename
- a name that does not contain a `/` is a host filename.

Filename validity

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

Output files

By default, the output files created by an ARM compiler are stored in the current directory. Object files are written in *ARM Executable and Linkable Format* (ELF). The ELF documentation is available in `*install_directory*\Pdf`.

2.2.2 Included files

Several factors affect the way the ARM compilers search for `#include` header files and source files. These include:

- the `-I` and `-j` compiler options
- the `-fk` and `-fd` compiler options
- the value of the environment variable *ARMINC*
- whether the filename is an absolute filename or a relative filename
- whether the filename is between angle brackets or double quotes.

The in-memory file system

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

The C++ header files that are equivalent to the C library header files are also stored in the in-memory file system. The header files specific to C++, such as `iostream`, are not stored in the in-memory file system.

Enclosing a filename in angle brackets, `#include <stdio.h>` for example, indicates that the included file is a system file and instructs the compiler to look in the in-memory file system first.

Enclosing a filename in double quotes, `#include "myfile.h"` for example, indicates that it is not a system file and instructs the compiler to look in the search path.

The current place

By default, the ARM compilers use Berkeley UNIX search rule, so source files and `#include` header files are searched for relative to the *current place*. This is the directory containing the source or header file currently being processed by the compiler.

When a file is found relative to an element of the search path, the directory containing that file becomes the new current place. When the compiler has finished processing that file, it restores the previous current place. At each instant there is a stack of current places corresponding to the stack of nested `#include` directives. For example, if the current place is *install_directory*`\include` and the compiler is seeking the include file `sys\defs.h`, it will locate *install_directory*`\include\sys\defs.h` if it exists.

When the compiler begins to process `defs.h`, the current place becomes *install_directory*`\include\sys`.

Any file included by `defs.h` that is not specified with an absolute pathname, is sought relative to *install_directory*`\include\sys`.

Only when the compiler has finished processing `defs.h` will the original current place *install_directory*`\include` be restored.

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

The ARMINC environment variable

You can set the `ARMINC` environment variable to a comma-separated list of directories in order to control searching for included header and source files. For example, from a command line, type:

```
set ARMINC=c:\work\x,c:\work\y
```

When compiling from the command line, directories specified with `ARMINC` will be searched immediately after directories specified by the `-I` option on the command line have been searched. If the `-j` option is used, `ARMINC` is ignored.

The search path

Table 2-2 shows how the various command-line options affect the search path used by the compiler when it searches for included header and source files. The following conventions are used in the table:

`:mem`	The in-memory file system where the ARM compilers store ANSI C and some C++ header files. See *The in-memory file system* on page 2-5 for more information.
`ARMINC`	The list of directories specified by the `ARMINC` environment variable, if it is set.
`CP`	The current place. See *The current place* on page 2-6 for more information.
`Idir and jdirs`	The directories specified by the `-I` and `-j` compiler options.

Table 2-2 Include file search paths

Compiler option	<include>	"include"
neither `-I` or `-j`	`:mem` and `ARMINC`	`CP`, `ARMINC`, and `:mem`
`-j`	`jdirs`	`CP` and `jdirs`
`-I`	`:mem`, `ARMINC`, and `Idirs`	`CP`, `Idirs`, `ARMINC`, and `:mem`
both `-I` and `-j`	`Idirs` and `jdirs`	`CP`, `Idirs`, and `jdirs`
`-fd`	No effect	Removes `CP` from the search path, so the search is now the same as that invoked with angle brackets.
`-fk`	No effect	Uses Kernighan and Ritchie search rules.

2.3 Command syntax

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

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

2.3.1 Invoking the compiler

The command for invoking the ARM compilers is:

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

The command-line options can appear in any order. The options are:

compiler	This is one of armcc, tcc, armcpp, or tcpp.
PCS-options	This specifies the procedure call standard to use. See *Procedure Call Standard options* on page 2-10 for details.
source-language	This specifies the variant of source language that is accepted by the compiler. The default is ANSI C for the C compilers and ISO Standard C++ for the C++ compilers. See *Setting the source language* on page 2-12 for details.
search-paths	This specifies the directories that are searched for included files. See *Specifying search paths* on page 2-13 for details.
preprocessor-options	This specifies preprocessor behavior, including preprocessor output and macro definitions. See *Setting preprocessor options* on page 2-14 for details.
output-format	This specifies the format for the compiler output. You can use these options to generate assembly language output listing files and object files. See *Specifying output format* on page 2-15 for details.
target-options	This specifies the target processor or architecture. See *Specifying the target processor or architecture* on page 2-17 for details.

`debug-options`	This specifies whether or not debug tables are generated, and their format. See *Generating debug information* on page 2-19 for details.
`code-generation-options`	This specifies options such as optimization, endianness, and alignment of data produced by the compiler. See *Controlling code generation* on page 2-20 for details.
`warning-options`	This specifies whether specific warning messages are generated. See *Controlling warning messages* on page 2-24 details.
`additional-checks`	This specifies several further checks that can be applied to your code, such as checks for data flow anomalies and unused declarations. See *Specifying additional checks* on page 2-29 for details.
`error-options`	This enables you to turn off specific recoverable errors or downgrade specific errors to warnings. See *Controlling error messages* on page 2-30 for details.
`source`	This provides the filenames of one or more text files containing C or C++ source code. By default, the compiler looks for source files, and creates output files, in the current directory.

Reading compiler options from a file

When the operating system restricts the command line length, use the following option to read additional command-line options from a file:

`-via` *`filename`*

This opens a file and reads additional command-line options from it. You can nest `-via` calls within via files by including `-via` *`filename2`* in the file.

In the following example, the options specified in `input.txt` are read as the command-line is parsed:

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

Specifying keyboard input

Use minus – as the source filename to instruct the compiler to take input from the keyboard. Input is terminated by entering `Ctrl-D` on UNIX environments or `Ctrl-Z` on MS Windows environments.

An assembly listing for the keyboard input is sent to the output stream at the end of each function if both of the following are true:

- no output file is specified
- no preprocessor-only option is specified, for example `-E`.

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

Getting help and version information

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

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

Redirecting errors

Use the `-errors` *`filename`* option to redirect compiler error output to a file. Errors on the command line are not redirected.

2.3.2 Procedure Call Standard options

This section applies to the *ARM/Thumb Procedure Call Standard* (ATPCS) as used by the ARM compilers.

See the *ADS Developer Guide* for more information on the ARM and Thumb procedure call standards. See *Controlling code generation* on page 2-20 for other build options.

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

```
-apcs qualifiers
```

The following rules apply to the `-apcs` command-line option:

- at least one qualifier must be present
- there must be no space between qualifiers.

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

```
-apcs /noswst/nointer/noropi/norwpi -fpu softVFP
```

The qualifiers are listed below.

Interworking qualifiers

`/nointerwork`
: This option compiles code with no ARM/Thumb interworking support. This is the default.

`/interwork`
: This option compiles code with ARM/Thumb interworking support. See the *ADS Developer Guide* for more information on ARM/Thumb interworking and Chapter 6 *The ARM Linker* for information on the automatically generated interworking veneers.

Position independence qualifiers

`/noropi`
: This option does not compile (read-only) position-independent code. This is the default. `/nopic` is an alias for this option.

`/ropi`
: This option compiles (read-only) position-independent code. `/pic`, for position independent code, is an alias for this option. If this option is selected the compiler:

- addresses read-only code and data pc-relative
- sets the *Position Independent* (PI) attribute on read-only output sections.

Note

The ARM tools cannot determine if the final output image will be *Read-Only Position Independent* (ROPI) until the linker finishes processing input sections. This means that the linker might emit ROPI error messages, even though you have selected this option.

`/norwpi`
: This option does not compile code that addresses read/write data position-independently. This is the default. `/nopid` is an alias for this option.

`/rwpi`
: This option compiles code that addresses read/write data position-independently. `/pid`, for position independent data, is an alias for this option. If this option is selected, the compiler:

- addresses writable data using offsets from the static base register sb. This means that:
  - data address can be fixed at runtime
  - data can be multiply instanced

  - — data can be, but does not have to be, position-independent.
- sets the PI attribute on read/write output sections.

Note

The compiler does not force your read/write data to be position-independent. This means that the linker might emit RWPI messages, even though you have selected this option.

Stack checking qualifiers

`/noswstackcheck`

This option does not use the software stack-checking *Procedure Call Standard* (PCS) variant. This is the default.

`/swstackcheck`

This option uses the software stack-checking PCS variant.

2.3.3 Setting the source language

This section describes options that determine the source language variant accepted by the compiler (see also *Controlling code generation* on page 2-20).

The following options specify how strictly the compiler enforces the standards and conventions of that language. By default, the C compilers compile ANSI-C, and the C++ compilers compile as much as they can of ISO/IEC C++.

`-ansi` This option compiles ANSI standard C. This is the default for armcc and tcc. The default mode is a fairly strict ANSI compiler, but without some of the inconvenient features of the ANSI standard. There are also some minor extensions allowed (for example `//` in comments and `$` in identifiers).

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

`-cpp` This option compiles ISO/IEC C++. This option is the default with the C++ compilers and not available with the C compilers.

`-embeddedcplusplus`

This option compiles standard *Embedded C++* (EC++). This option is not available with the C compilers.

`-strict`	This option enforces more stringent conformance to the ANSI C standard and the ISO/IEC C++ standard. For example, the following code: `static struct T {int i; };` gives an error when compiled with `-cpp -strict`, but only a warning with `-cpp`. Because no object is declared, `static` is spurious. In the C++ standard, the code shown is therefore illegal.

It is possible to combine language options:

`armcc -ansi`	Compiles ANSI standard C. This is the default.
`armcc -strict`	Compiles strict ANSI standard C.
`armcpp`	Compiles standard C++.
`armcpp -ansi`	Compiles normal ANSI standard C. (C mode of C++).
`armcpp -ansi -strict`	Compiles strict ANSI standard C. (C mode of C++).
`armcpp -strict`	Compiles strict C++.

2.3.4 Specifying search paths

The following options specify the directories that are searched for included files.

The precise search path will vary according to the combination of options selected and whether the include file is enclosed in angle brackets or double quotes. See *Included files* on page 2-5 for full details of how these options work together.

`-I`*`dir-name`*	This option adds the specified directory to the list of places that are searched for included files. If more than one directory is specified, the directories are searched in the same order as the `-I` options specifying them. ARM uses an in-memory file system to speed processing of include header files. The in-memory file system is specified by `-I-`.
`-fk`	This option uses Kernighan and Ritchie search rules for locating included files. The current place is defined by the original source file and is not stacked. See *The current place* on page 2-6 for more information. If you do not use this option, Berkeley-style searching is used.

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

`-j`*`dir-list`* This option adds the specified comma-separated list of directories to the end of the search path after all the directories specified by the `-I` options. Use `-j-` to search the in-memory file system.

2.3.5 Setting preprocessor options

The following command-line options control aspects of the preprocessor. (See *Pragmas* on page 3-2 for descriptions of other preprocessor options that can be set by pragmas.)

`-E` This option executes only the preprocessor phase of the compiler. By default, output from the preprocessor is sent to the standard output stream and can be redirected to a file using standard UNIX and MS-DOS notation. For example:

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

You can also use the `-o` option to specify a file for the preprocessed output. If `-E` is specified without `-o`, output is sent to the standard output stream. By default, comments are stripped from the output. See also the `-C` option.

`-C` This option retains comments in preprocessor output when used in conjunction with `-E`. This option differs from the `-c` (lowercase) option that suppresses the link step. See *Specifying output format* on page 2-15 for a description of the `-c` option.

`-D`*`symbol=value`*

This option defines *`symbol`* as a preprocessor macro. This has the same effect as the text `#define` *`symbol value`* at the head of the source file. This option can be repeated.

`-D`*`symbol`* This option defines *`symbol`* as a preprocessor macro. This has the same effect as the text `#define` *`symbol`* at the head of the source file. This option can be repeated. The default value of symbol is 1.

`-M` This option executes only the preprocessor phase of the compiler, as with `-E`. This option produces a list of makefile dependency lines suitable for use by a **`make`** utility. By default, output is on the standard output stream. You can redirect output to a file by using standard UNIX and MS-DOS notation. For example:

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

If the `-o` *`filename`* option is specified, the dependency lines generated on standard output refer to `filename.o`, not to `source.o`. However, no object file is produced with the combination of `-M -o` *`filename`*.

`-U`*`symbol`* This option undefines *`symbol`*. This has the same effect as the text `#undef` *`symbol`* at the head of the source file. This option can be repeated.

2.3.6 Specifying output format

By default, source files are compiled and linked into an executable image.

Use the following options to direct the compiler to create unlinked object files, assembly language files, or listing files from C or C++ source files.

`-c` This option compiles but does not perform the link step. The compiler compiles the source program and writes the object files to either the current directory or the file specified by the `-o` option. This option is different from the uppercase `-C` option, described in *Setting preprocessor options* on page 2-14. (The `-C` option retains comments in preprocessor output.)

`-list` This option creates a listing file consisting of lines of source interleaved with error and warning messages. The options `-fi`, `-fj`, and `-fu` can be used to control the contents of this file.

Caution

The `-list` option does not accept a pathname for the output file. You must rename previous versions of list files if you do not want to overwrite them.

`-fi` This option is used with `-list` to list the lines from any files included with directives of the form `#include "file"`.

`-fj` This option is used with `-list` to list the lines from any files included with directives of the form `#include <file>`.

`-fu` This option is used with `-list` to list source that was not preprocessed.

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

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

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

```
p = 0;
```

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

```
p = NULL;
```

`-o` *`file`* This option names the file that holds the final output of the compilation:

- if *file* is `-`, the output is written to the standard output stream and `-S` is assumed (unless `-E` is specified).
- used with `-c`, it names the object file.
- used with `-S`, it names the assembly language file.
- used with `-E`, it specifies the output file for preprocessed source.
- if none of `-c` , `-S`, or `-E` is present, it specifies the output file of the link step. An executable image called `file` is created.

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

`-MD` This option compiles the source and writes makefile dependency lines to file *`inputfilename`*`.d`. The output file is suitable for use by a **`make`** utility.

`-depend` *`filename`*

This option is the same as `-MD`, but writes makefile dependency lines to the specified file.

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

Note

You can use armasm to assemble the output file and produce object code. You must, however, specify the same ATPCS settings in the assembler as were used in the compiler. The software stack check (`/swst`) default for the assembler is not the same as the default for the compiler.

`-fs` This option, when used with `-S`, interleaves C, or C++, source code line by line as comments within the compiler-generated assembler code.

2.3.7 Specifying the target processor or architecture

The options described in this section specify the target processor or architecture attributes for a compilation. The compiler can take advantage of certain extra features of the selected processor or architecture, such as support for halfword load and store instructions and instruction scheduling.

Note

Specifying the target processor can make the code incompatible with other ARM processors.

The following general points apply to processor and architecture options:

- The supported `-cpu` values are all current ARM product names or architecture versions. There are no aliases or wildcard matching.
- If you specify an architecture name for the `-cpu` option, the code is compiled to run on any processor supporting that architecture, for example `-cpu 4T` produces code that can be used by either the ARM7TDMI or ARM9TDMI.
- If you specify a processor for the `-cpu` option, for example `-cpu ARM940T`, the compiled code will be optimized for that processor. This allows the compiler to use specific coprocessors or instruction scheduling for optimum performance.
- Use only a single processor or architecture name with `-cpu`. You cannot specify both a processor and an architecture.
- If `-cpu` is not specified, the default is `-cpu ARM7TDMI`.
- Specifying a Thumb-aware processor, such as `-cpu ARM7TDMI` to armcc or armcpp does not make these compilers generate Thumb code. It only allows features of the processor to be used, such as interworking instructions. Use tcc or tcpp to generate Thumb code.

The following options are available:

`-cpu` *`name`* This option compiles code for a specific ARM processor or architecture.

If *`name`* is a processor:

- The name must be entered exactly as it is shown on ARM data sheets, for example `ARM7TDMI`. Wildcard characters are not accepted. Valid values are any ARM-supported processor later than ARM6.
- The selection of the processor will select the appropriate architecture, fpu, and memory organization.
- The `-cpu` selection implies `-fpu`. The implied `-fpu` can be overridden by an explicit `-fpu` *`option`*. Where no FPU is available, `-fpu softvfp` is used.

If *`name`* is an architecture, it must be one of:

`3`	ARMv3 without long multiply
`3M`	ARMv3 with long multiply
`4`	ARMv4 with long multiply but no Thumb
`4xM`	ARMv4 without long multiply or Thumb
`4T`	ARMv4 with long multiply and Thumb
`4TxM`	ARMv4 without long multiply but with Thumb
`5T`	ARMv5 with long multiply and Thumb.

`-fpu` *`name`* This option selects the target *floating-point unit* (FPU) architecture, where *`name`* is one of:

`none`	Selects no floating-point option. No floating-point code is to be used.
`VFP`	Selects hardware Vector Floating Point unit.
`FPA`	Selects hardware Floating Point Accelerator.
`softVFP`	Selects software *floating-point library* (FPLib) with pure-endian doubles.
`softFPA`	Selects software floating-point library with mixed-endian doubles.

2.3.8 Generating debug information

There are options that enable you to specify whether debug tables are generated for the current compilation and, if they are, specify their format. See *Pragmas* on page 3-2 for more information on controlling debug information.

Note

Optimization criteria can limit the debug information generated by the compiler. See *Defining optimization criteria* on page 2-20 for more information.

Debug table generation options

The following options specify how debug tables are generated:

`-g [options]`

This option switches on the generation of debug tables for the current compilation. Debug table options are as specified by `-gt`. The compiler produces the same code whether `-g` is used or is not used. The only difference is the existence of debug tables.

Optimization options for debug code are specified by `-O`. By default, the `-g` option on its own is equivalent to:

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

Debug data is not generated for **`inline`** functions unless `-Ono_inline` is used.

`-g+` This is a synonym for `-g`. It is generated by graphical configurers (the CodeWarrior IDE for example).

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

`-gt[p]` This option, when used with `-g`, specifies the debug table entries that generate source level objects.

Debug tables can be very large, so it is sometimes useful to limit their size by restricting what is included:

`-gt` Generates all available entries. This is the default option.

`-gtp` Prevents preprocessor macro definitions in debug tables. This option is ignored if DWARF1 debug tables are generated, because there is then no way to describe macros.

Debug table format options

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

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

`-dwarf1` This option specifies DWARF1 debug table format. This option is not recommended for C++. Specify `-dwarf2` instead of `-dwarf1`. This option will not be supported in the future.

2.3.9 Controlling code generation

Use the options described in this section to control aspects of the code generated by the compiler such as optimization. See *Pragmas* on page 3-2 for information on additional code generation options that are controlled using pragmas.

This section describes:

- *Defining optimization criteria* on page 2-20
- *Controlling code and data sections* on page 2-22
- *Setting byte order* on page 2-22
- *Setting alignment options* on page 2-22.

Defining optimization criteria

`-Ono_inline`

This option disables inlining. Calls to inline functions are not expanded inline. This option is not available with `-dwarf1`.

`-Oinline` This option expands inline functions instead of placing them in a common code section. Inline functions are difficult to debug if this option is used. This is the default.

`-Ospace` This option optimizes to reduce image size at the expense of a possible increase in execution time. For example, large structure copies are done by out-of-line function calls instead of inline code. Use this option if code size is more critical than performance. This is the default.

`-Otime` This option optimizes to reduce execution time at the possible expense of a larger image. Use this option if execution time is more critical than code size. For example, it compiles:

```
while (expression) body;
```

as:

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

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

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

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

`-O1` Turns off the following optimizations:

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

If used with `-g`, this option gives a satisfactory debug view with good code density.

`-O2` Generates fully optimized code. If used with `-g+`, the debug view might be less satisfactory since the mapping of object code to source code is not always clear. This is the default optimization level if debug tables are not generated.

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

- For all debug table formats, if optimization level 1 or 2 is used, the debugger can display misleading values for local variables. If a variable is not live at the point where its value is interrogated, its location can be used for some other variable. In addition, the compiler replaces some local variables with constant values at each place the variable is used.
- For all debug table formats, if optimization level 2 is used, the value of variables that have been subject to range splitting or structure splitting cannot be displayed.

See *Pragmas* on page 3-2 for more information on controlling optimization.

Controlling code and data sections

`-zo`	This option generates one ELF section for each function in source file. This option enables the linker to remove unused functions when the default `-remove` linker option is active. This option increases code size slightly for some functions, but when creating code for a library, it can prevent unused functions being included at the link stage. This can result in the reduction of the final image size.

Setting byte order

`-littleend`	This option compiles code for an ARM processor using little-endian memory. With little-endian memory, the least significant byte of a word has lowest address. This is the default.
`-bigend`	This option compiles code for an ARM processor using big-endian memory. With big-endian memory, the most significant byte of a word has lowest address.

Setting alignment options

`-zas`*`Number`*	This option specifies the minimum byte alignment for structures. Allowed values for *`Number`* are: `1, 2, 4, 8` The default is 1. This option is deprecated and will not be supported in future versions of the product.

`-memaccess` *option*

This option indicates to the compiler that the memory in the target system has slightly restricted or expanded capabilities. By default, ARM compilers assume that the memory system can load and store words at 4-byte alignment, halfwords at 2-byte alignment, and bytes. Load and store capability can be indicated by specifiying *option*:

`+L41`	The memory can return the aligned word containing the addressed byte. This is useful only with architecture v3 processors that lack load halfword.
`-S22`	The memory cannot store halfwords. This can be used to suppress the generation of `STRH` instructions when generating ARM code for architecture v4 (and later) processors.
`-L22`	The memory cannot load halfwords. This can be used to suppress the generation of `LDRH` instructions when generating ARM code for architecture v4 (and later) processors.

Note

Do not use `-L22` or `-S22` when producing Thumb code.

It is possible that the processor has memory access modes available that the physical memory lacks (load aligned halfword for example).

It is also possible that the physical memory has access modes that the processor cannot use (architecture v3 load aligned halfword for example).

Controlling implementation details

`-fy` This option forces all enumerations to be stored in integers. This option is switched off by default and the smallest data type is used that can hold the values of all enumerators.

Note

This option is not recommended for general use and is not required for ANSI-compatible source.

`-zc` This option makes the **`char`** type to be signed. It is normally unsigned in C++ and ANSI C modes.

Note

This option is not recommended for general use and is not required for ANSI-compatible source. If used incorrectly, this option can cause errors in the resulting image.

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

2.3.10 Controlling warning messages

The compiler issues warnings about potential portability problems and other hazards. The compiler options allow you to turn off specific warnings. For example, you can turn off warnings if you are in the early stages of porting a program written in old-style C. In general, it is better to check the code than to switch off warnings.

The options are on by default, unless specified otherwise.

See also *Specifying additional checks* on page 2-29 for descriptions of additional warning messages.

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

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

where the *`options`* field contains zero or more characters.

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

You can specify several options at the same time. For example:

```
-Wad+fg
```

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

The warning message options are as follows:

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

`-Wa` This option suppresses the warning:

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

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

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

where it is possible that one of the following was intended:

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

`-Wb` This option suppresses the warning messages that are issued for extensions to the ANSI standard. Examples include:

- using an unwidened type in an ANSI C assignment
- specifying bitfields with a type of **`char`**, **`short`**, **`long`**, or **`long long`**
- specifying **`char`**, **`short`**, **`float`**, or **`enum`** arguments to variadic functions such as `va_start()`.

`-Wd` This option suppresses the warning message:

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

This warning is normally given when a declaration without argument types is encountered in ANSI C mode.

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

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

`-We` This option suppresses warnings about pointer casts in static int initializations.

`-Wf` This option suppresses the message:

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

This is an error in C++ and cannot be suppressed. It is a warning in ANSI C and suppressing this message can be useful when compiling old-style C in ANSI C mode.

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

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

-Wi
: This option suppresses the implicit constructor warning (C++ only). It is issued when the code requires a constructor to be invoked implicitly. For example:

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

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

-Wl
: This option gives the warning message:

```
Lower precision in wider context
```

when code like the following is found:

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

where the multiplication yields an **int** result that is then widened to **long**. This warning indicates a potential problem when either the destination is **long long** or where the code has been ported to a system that uses 16-bit integers or 64-bit longs. This option is off by default. It can be enabled with -W+l.

-Wm
: This option suppresses warnings about multiple-character char constants.

-Wn
: This option suppresses the warning message:

```
Implicit narrowing cast
```

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

Such implicit narrowing casts are almost always a source of problems when moving code that has been developed on a fully 32-bit system to a system where **int** occupies 16 bits and **long** occupies 32 bits. The -Wn option is suppressed by default.

`-Wo` This option suppresses warnings for implicit conversion to signed **`long long`** constants.

`-Wp` This option suppresses the warning message:

```
non-ANSI #include <…>
```

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

`-Wq` This option suppresses warnings in C++ constructor initialization order.

`-Wr` This option suppresses the implicit virtual warning (C++ only) issued when a non-virtual member function of a derived class hides a virtual member of a parent class. For example:

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

Adding the **`virtual`** keyword in the derived class prevents the warning.

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

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

`-Wu` For C code, `-Wu` suppresses warnings about future compatibility with C++. Warnings are suppressed by default. They can be enabled with `-W+u`.

`-Wv` This option suppresses the warning message:

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

This is usually caused by a return from a function that was assumed to return **`int`**, because no other type was specified, but is being used as a void function. This is widespread in old-style C mode. Such action will always result in an error in C++.

`-Wx` This option suppresses unused declaration warnings such as:

```
C2870W: variable 'y' declared but not used
```

By default, unused declaration warnings are given for:

- Local (within a function) declarations of variables, typedefs, and functions
- Labels (always within a function)
- Top-level static functions and static variables.

`-Wy` This option turns off warnings about deprecated features.

2.3.11 Specifying additional checks

The options described below give you control over the extent and rigor of the checks. Additional checking is an aid to portability and is good coding practice.

`-fa` This option checks for certain types of data flow anomalies. The compiler performs data flow analysis as part of code generation. The checks indicate when an automatic variable might have been used before being assigned a value. The check is pessimistic and will sometimes report an anomaly where there is none. In general, it is useful at some stage to check all code using `-fa`.

`-fh` This option checks that:

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

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

If external objects are declared only in included header files and are never inline in a source file, these checks directly support good modular programming practices.

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

`-fp` This option reports on explicit casts of integers to pointers, for example:

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

This warning indicates potential portability problems. Casting explicitly between pointers and integers, although not clean, is not harmful on the ARM processor where both are 32-bit types. This option also causes casts to the same type to produce a warning. For example:

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

`-fv`	This option reports on all unused declarations (including from standard headers).
`-fx`	This option enables all warnings normally suppressed by default.

2.3.12 Controlling error messages

The compiler issues errors to indicate serious problems in the code it is attempting to compile. The compiler options described below allow you to:

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

——— **Caution** ———

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

These options can be useful during development, or when importing code from other environments. However, they might allow code to be produced that does not function correctly. It is generally better to correct the code than to use options to switch off error messages.

———————————

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

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

where *`options`* is a set of one or more of the letters `a`, `c`, `f`, `i`, `l`, `p`, or `z` as described below.

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

——— **Note** ———

The `-E` option on its own without any options is the preprocessor switch. See *Setting preprocessor options* on page 2-14.

———————————

You can specify multiple options. For example:

```
-Eac
```

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

The following options are on by default:

`-Ea` For C++ only, this option downgrades access control errors to warnings. For example:

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

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

`-Ef` This option suppresses errors for unclean casts, such as **`short`** to **`pointer`**.

`-Ei` For C++ only, this option downgrades from error to warning the use of implicit **`int`** in constructs such as:

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

`-El` This option suppresses errors about linkage disagreements where functions are implicitly declared as **`extern`** and then later redeclared as **`static`**.

`-Ep` This option suppresses errors arising as the result of extra characters at the end of a preprocessor line.

`-Ez` This option suppresses the errors caused by zero-length arrays.

Chapter 3
ARM Compiler Reference

This chapter contains reference information for the ARM compilers. It contains the following sections:

-

3.1 Compiler-specific features

This section describes the ARM-specific aspects of the ARM C and C++ compilers, including:

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

Note

Features described here are outside the ANSI specification and might not easily port to other compilers.

3.1.1 Pragmas

Pragmas of the following form are recognized by the ARM compiler:

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

Pragmas are listed in Table 3-1. The following sections describe these pragmas in more detail.

Table 3-1 Pragmas recognized by the ARM compilers

Pragma name	Default	Reference
`check_printf_formats`	Off	*Pragmas controlling printf/scanf argument checking* on page 3-3
`check_scanf_formats`	Off	*Pragmas controlling printf/scanf argument checking* on page 3-3
`check_stack`	On	*Pragmas controlling code generation* on page 3-4
`debug`	On	*Pragmas controlling debugging* on page 3-3
`Ospace`	–	*Pragmas controlling optimization* on page 3-3
`Otime`	–	*Pragmas controlling optimization* on page 3-3
`O`*n*	–	*Pragmas controlling optimization* on page 3-3
`softfp_linkage`	Off	*Pragmas controlling code generation* on page 3-4

-

Pragmas controlling printf/scanf argument checking

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

`check_printf_formats`

This pragma marks printf-like functions for type checking against a literal format string, if it exists. If the format is not a literal string, no type checking is done. The format string must be the last fixed argument. For example:

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

`check_scanf_formats`

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

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

Pragmas controlling debugging

The following pragma controls aspects of debug table generation:

`debug` This pragma turns debug table generation on or off.

If `#pragma no_debug` is specified, no debug table entries are generated for subsequent declarations and functions until the next `#pragma debug`.

Pragmas controlling optimization

The following pragmas control aspects of optimization:

`Ospace`	This pragma optimizes for space (capital letter O).
`Otime`	This pragma optimizes for time.
`O`*`num`*	This pragma changes optimization level. The value of *num* is 0, 1, or 2.

Pragmas controlling code generation

The following pragmas control how code is generated. Many other code generation options are available from the compiler command line.

`check_stack`

This pragma reenables the generation of function entry code that checks for stack limit violation if the `-noswst` (default) command-line option is used.

`softfp_linkage`

This pragma asserts that all function declarations up to the next `#pragma no_softfp_linkage` describe functions that use software floating-point linkage. The **`__softfp`** keyword has the same effect and should be used instead (see *Function declaration keywords* on page 3-4). The pragma form can be useful when applied to an entire interface specification (header file) without altering that file.

3.1.2 Function declaration keywords

Several function declaration keywords tell the compiler to give a function special treatment. These are all ARM extensions to the ANSI C specification.

`__asm` This instructs the compiler that the following code is written in assembler language (see *Inline assembler* on page 3-35).

`__inline` This instructs the compiler to compile C functions inline. The semantics of `__inline` are exactly the same as those of the C++ **`inline`** keyword:

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

The compiler compiles functions inline when `__inline` is used and the functions are not too large. Large functions are not compiled inline because they can adversely affect code density and performance.

`__irq` This enables a C or C++ function to be used as an interrupt routine called via the `IRQ` or `FIQ` vectors. All corrupted registers (not just those normally preserved under the ATPCS) except floating-point registers, are preserved. The default ATPCS mode must be used. The function exits by setting the pc to lr–4 and the CPSR to the value in SPSR. It is not available in tcc or tcpp. No arguments or return values can be used with `__irq` functions.

Refer to Handling Processor Exceptions in the *ADS Developer Guide* for detailed information on using `__irq`.

-

`__pure` This asserts that a function declaration is pure. Functions that are pure are candidates for common subexpression elimination. By default, functions are assumed to be impure (causing side-effects). A function is properly defined as pure only if:

- its result depends exclusively on the values of its arguments
- it has no side effects, for example it cannot call impure functions.

So, a pure function cannot use global variables or dereference pointers, because the compiler assumes that the function does not access memory (except stack memory) at all. When called twice with the same parameters, a pure function must return the same value each time.

`__softfp` This asserts that a function uses software floating-point linkage. Calls to the function pass floating-point arguments in integer registers. If the result is floating-point, the value to be returned in an integer register. This duplicates the behavior of compilation targeting software floating-point.

This keyword allows an identical library to be used by sources compiled to use hardware and software floating-point.

`__swi` This declares a SWI function taking up to four integer-like arguments and returning up to four results in a `value_in_regs` structure. This causes function invocations to be compiled inline as an ATPCS compliant SWI that behaves similarly to a normal call to a function.

For a SWI returning no results, use:

```
void __swi(swi_num) swi_name(int arg1,…, int argn);
```

For example:

```
void __swi(42) terminate_proc(int procnum);
```

For a SWI returning one result, use:

```
int __swi(swi_num) swi_name(int arg1,…, int argn);
```

For a SWI returning more than 1 result use:

```
typedef struct res_type { int res1,…,resn;} res_type;
res_type __value_in_regs __swi(swi_num) swi_name(
           int arg1,…,int argn);
```

The `__value_in_regs` qualifier is used to specify that a small structure of up to four words (16 bytes) is returned in registers, rather than by the usual structure-passing mechanism defined in the ATPCS.

Refer to Handling Processor Exceptions in the *ADS Developer Guide* for detailed information.

`__swi_indirect`

This passes an operation code to the SWI handler in r12:

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

where:

swi_num	Is the SWI number used in the SWI instruction.
real_num	Is the value passed in r12 to the SWI handler. This feature can be used to implement indirect SWIs. The SWI handler can use r12 to determine the function to perform.

For example:

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

This SWI can be called as follows:

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

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

To use the indirect SWI mechanism, your system SWI handlers must make use of the r12 value to select the required operation.

`__value_in_regs`

This instructs the compiler to return a structure of up to four integer words in integer registers or up to four floats or doubles in floating-point registers rather than using memory, for example:

```
typedef struct int64_struct {
    unsigned int lo;
    unsigned int hi;
} int64_struct;

__value_in_regs extern
    int64_struct mul64(unsigned a, unsigned b);
```

Declaring a function `__value_in_regs` can be useful when calling assembler functions that return more than one result. See the *ADS Debuggers Guide* for information on the default method of passing and returning structures.

Note

A C++ function cannot return a `__value_in_regs` structure if the structure requires copy constructing.

`__weak` This specifies an **`extern`** function or object declaration that, if not present, does not cause the linker to fault an unresolved reference. The linker will not load the function or object from a library unless another compilation uses the function or object non-weakly. If the reference remains unresolved, its value is assumed to be `NULL`.

If the reference is made from code that compiles to a Branch or Branch-Link instruction, the reference is resolved as branching to the next instruction. This effectively makes the branch a no-op.

```
__weak void f(void);
...
f(); // call f weakly
```

A function or object cannot be used both weakly and non-weakly. For example the following code uses `f()` weakly from `g()` and `h()`.

```
void f(void);
void g() {f();}
__weak void f(void);
void h() {f();}
```

It is not possible to use a function or object weakly from the same compilation that defines the function or object. The code below uses `f()` non-weakly from `h()`.

```
__weak void f(void);
void h() {f();}
void f() {}
```

3.1.3 Variable declaration keywords

This section describes the implementation of ARM-specific variable declaration keywords.

Standard keywords

Standard C keywords are described in *Standard C implementation definition* on page 3-11. For examples, see:

- *Qualifiers* on page 3-26 for information on qualifiers such as **`volatile`** and `__packed`
- *Registers* on page 3-19 for information on the standard keyword **`register`**.

ARM-specific keywords

The keywords in this section are used to declare or modify variable definitions:

`__int64` This is an alternative name for type **`long long`**. This is accepted even when using `-strict`

`__global_reg(`*`vreg`*`)`

This allocates the declared variable to a global integer register variable. *vreg* is an ATPCS callee-save register (v1) and not a real register number (r1). Global register variables cannot be qualified or initialized at declaration. Valid types are:

- any integer type, except **`long long`**
- any pointer type.

For example, to declare a global integer register variable allocated to r5 (the ATPCS register v2), use the following:

```
__global_reg(2) int x;
```

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

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

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

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

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

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

——— **Caution** ————

Exercise care when using global register variables due to the following:

- There is no check at link time to ensure that direct calls between different compilation units are sensible. If possible, any global register variables used in a program should be defined in each compilation unit of the program. In general, it is best to place the definition in a global header file. Your code must set up the value in the global register early in your code before the register is used.
- A global register variable maps to a callee-saved register, so its value is saved and restored across a call to a function in a compilation unit that does not use it as a global register variable, such as a library function.
- Calls back into a compilation unit that uses a global register variable are dangerous. For example, if a global register using function is called from a compilation unit that does not declare the global register variable, the function will read the wrong values from its supposed global register variables.

Qualifiers

See *Qualifiers* on page 3-26 for information on the `__packed` and `volatile` qualifiers.

3.1.4 Size and alignment of basic data types

Table 3-2 gives the size and natural alignment of the basic data types. Type alignment varies according to the context. (See *Structures, unions, enumerations, and bitfields* on page 3-20.)

- Local variables usually kept in registers, but when local variables are spilled onto the stack, they are always word-aligned. For example, a spilled local **`char`** variable has an alignment of 4.
- The natural alignment of a packed type is 1.

Table 3-2 Size and alignment of data types

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

3.2 Standard C implementation definition

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

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

- *Nonconformance with ANSI* on page 3-11
- *Translation* on page 3-12
- *Environment* on page 3-12
- *Identifiers* on page 3-14
- *Characters* on page 3-15
- *Integers* on page 3-17
- *Floating-point* on page 3-18
- *Arrays and pointers* on page 3-19
- *Registers* on page 3-19
- *Structures, unions, enumerations, and bitfields* on page 3-20
- *Qualifiers* on page 3-26
- *Declarators* on page 3-28
- *Statements* on page 3-28
- *Preprocessing directives* on page 3-29
- *Library functions* on page 3-29.

Note

This section does not duplicate information that is part of the compiler-specific implementations. See *Compiler-specific features* on page 3-2. This section provides references where applicable.

3.2.1 Nonconformance with ANSI

The compiler behavior differs from the behavior described in the language conformance sections of the C standard in that there is no support for the `wctype.h` and `wchar.h` headers.

3.2.2 Translation

Diagnostic messages produced by the compiler are of the form:

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

where *`severity`* is one of:

`Warning` This is a helpful message from the compiler relating to a minor violation of the ANSI specification.

`Error` This is a violation of the ANSI specification but the compiler is able to recover by guessing the intention.

`Serious error`
This is a violation of the ANSI specification and no recovery is possible because the intention is not clear.

`Fatal error`
This is an indication that the compiler limits have been exceeded, or that the compiler has detected an internal fault (for example, not enough memory).

`error-code` is a number identifying the error type.

`explanation` is a text description of the error.

3.2.3 Environment

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

- *main()*
- *Interactive device* on page 3-13
- *Standard input, output and error streams* on page 3-13.

main()

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

Note

A whitespace character is any character where the result of `isspace()` is true.

A double quote or backslash character \ inside double quotes must be preceded by a backslash character.

An input/output redirection will not be recognized inside double quotes.

Interactive device

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

- No buffering is done on any stream connected to `:tt` unless input/output redirection has occurred.
- If input/output redirection other than to `:tt` has occurred, full file buffering is used (except that line buffering is used if both `stdout` and `stderr` were redirected to the same file).

Standard input, output and error streams

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

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

and redirects the files as follows:

`stdin`	The file is redirected to `infile`
`stdout`	The file is redirected to `outfile`
`stderr`	The file is redirected to `errfile`.

The permitted redirections are:

`0<` *`filename`*	This reads `stdin` from *`filename`*
`<` *`filename`*	This reads `stdin` from *`filename`*

`1>` *`filename`*	This writes `stdout` to *`filename`*
`>` *`filename`*	This writes `stdout` to *`filename`*
`2>` *`filename`*	This writes `stderr` to *`filename`*
`2>&1`	This writes `stderr` to the same place as `stdout`
`>&` *`filename`*	This writes both `stdout` and `stderr` to *`filename`*
`>>` *`filename`*	This appends `stdout` to *`filename`*
`>>&` *`filename`*	This appends both `stdout` and `stderr` to *`filename`*

File redirection is done only if either:

- the invoking operating system supports it
- the program reads and writes characters and has not replaced the C library functions `fputc()` and `fgetc()`.

3.2.4 Identifiers

The following points apply to the identifiers expected by the compilers:

- An identifier can be any length. The compiler truncates an identifier after 256 characters — all 256 characters are significant.
- Uppercase and lowercase characters are distinct in all internal and external identifiers. An identifier can also contain a dollar `($)` character unless the `-strict` compiler option is specified.

3.2.5 Characters

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

- Uppercase and lowercase characters are distinct in all internal and external identifiers. An identifier can also contain a dollar `($)` character unless the `-strict` compiler option is specified.
- Calling `setlocale(LC_CTYPE, "ISO8859-1")` makes the `isupper()` and `islower()` functions behave as expected over the full 8-bit Latin-1 alphabet, rather than over the 7-bit ASCII subset. The locale must be selected at link-time. (See *Tailoring locale and CTYPE* on page 4-24).
- 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 can appear in a string or character constant, and in a comment.
- The ARM compilers do not support multibyte character sets, for example Unicode.
- Other properties of the source character set are host-specific.

The properties of the execution character set are target-specific. The ARM C and C++ libraries support the ISO 8859-1 (Latin-1 Alphabet) character set with the following consequences:

- The execution character set is identical to the source character set.
- There are eight bits in a character in the execution character set.
- There are four characters (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.

- In C all character constants have type **`int`**. In C++ a character constant containing one character has the type **`char`** and 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 last character in the constant occupies the lowest-order byte of the integer value. Up to three preceding characters are placed at higher-order bytes. Unused bytes are filled with the `NULL` (`\0`) character.

- All integer character constants that contain a single character, or character escape sequence (see Table 3-3 on page 3-16), are represented in both the source and execution character sets.
- Characters of the source character set in string literals and character constants map identically into the execution character set.
- Data items of type **`char`** are unsigned by default. They can be explicitly declared as **`signed char`** or **`unsigned char`**. The `-zc` option can be used to make the **`char`** signed.
- No locale is used to convert multibyte characters into the corresponding wide characters (codes) for a wide character constant. This is not relevant to the generic implementation.

Table 3-3 Character escape codes

Escape sequence	Char value	Description
`\a`	`7`	attention (bell)
`\b`	`8`	backspace
`\t`	`9`	horizontal tab
`\n`	`10`	new line (line feed)
`\v`	`11`	vertical tab
`\f`	`12`	form feed
`\r`	`13`	carriage return
`\xnn`	`0xnn`	ASCII code in hexadecimal
`\nnn`	`0nnn`	ASCII code in octal

3.2.6 Integers

Integers are represented in two’s complement form. The low word of a **`long long`** is at the low address in little-endian mode, and at the high address in big-endian mode.

Operations on integral types

The following statements apply to operations on the integral types:

- All signed integer arithmetic uses a two's complement representation.
- Bitwise operations on signed integral types follow the rules that arise naturally from two's complement representation. No sign extension takes place.
- Right shifts on signed quantities are arithmetic.
- Any quantity that specifies the amount of a shift is treated as an unsigned 8-bit value.
- Any value to be shifted is treated as a 32-bit value.
- Left shifts of more than 31 give a result of zero.
- Right shifts of more than 31 give a result of zero from a shift of an unsigned value or positive signed value. They yield –1 from a shift of a negative signed value.
- The remainder on integer division has the same sign as the divisor.
- If a value of integral type is truncated to a shorter signed integral type, the result is obtained by discarding an appropriate number of most significant bits. If the original number was too large, positive or negative, for the new type, there is no guarantee that the sign of the result will be the same as the original.
- A conversion between integral types does not raise an exception.
- Integer overflow does not raise an exception.
- Integer division by zero raises a SIGFPE exception.

3.2.7 Floating-point

Floating-point quantities are stored in IEEE format:

- **`float`** values are represented by IEEE single-precision values
- **`double`** and **`long double`** values are represented by IEEE double-precision values.

If `softvfp` or `vfp` is selected, **`double`** and **`long double`** quantities, the word containing the sign, the exponent, and the most significant part of the mantissa is stored with the lower machine address in big-endian mode and at the higher address in little-endian mode. Refer to *Operations on floating-point types* on page 3-18 for more information.

ARM implements an ANSI extension for floating-point constants (see *Hexadecimal floating-point constants* on page 3-36).

Operations on floating-point types

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

- normal IEEE 754 rules apply
- rounding is to the nearest representable value by default
- conversion from a floating-point type to an integral type causes a floating-point exception to be raised only if the value cannot be represented in the destination type (**`int`** or **`long long`**)
- floating-point underflow is disabled by default
- floating-point overflow raises a SIGFPE exception by default
- floating-point divide by zero raises a SIGFPE exception by default.

——— **Caution** ———

The IEEE 754 standard for floating-point processing states that the default action to an exception is to proceed without a trap. You can modify fp error handling by tailoring the functions and definitions in `fenv.h`. See *Tailoring error signalling, error handling, and program exit* on page 4-47 and the chapter on floating-point in the *ADS Developer Guide*.

3.2.8 Arrays and pointers

The following statements apply to all pointer types in C. They also apply to all pointer types, except pointers to members, in C++:

- adjacent bytes have addresses that differ by one
- the macro `NULL` expands to the value 0
- casting between integers and pointers results in no change of representation
- the compiler warns of casts between pointers to functions and pointers to data
- the type `size_t` is defined as `unsigned int`
- the type `ptrdiff_t` is defined as `signed int`.

Pointer subtraction

The following statements apply to all pointers in C. They also apply to pointers, other than pointers to members, in C++:

- When one pointer is subtracted from another, the difference is obtained as if by the expression:

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

- If the pointers point to objects whose size is one, two, or four bytes, the natural alignment of the object ensures that the division will be exact, provided the objects are not packed.

- For packed or longer types, such as **`double`** and **`struct`**, the division might not be exact unless both pointers are to elements of the same array. Also, the quotient might be rounded up or down at different times, or in different circumstances. This can lead to inconsistencies.

3.2.9 Registers

Using the ARM compilers, you can declare any number of local objects (auto variables) to have the storage class **`register`**.

Depending on the variant of the ATPCS being used, there are between five and seven integer registers available, and four floating-point registers. In general, declaring more than four integer register variables and two floating-point register variables is not recommended.

-

The following object types can be declared to have the **`register`** storage class:

- All integer types (long long occupies two registers).
- All integer-like structures. That is, any one word **`struct`** or **`union`** where all addressable fields have the same address, or any one word structure containing bitfields only. The structure must be padded to 32 bits.
- Any pointer type.
- Floating-point types. The double precision floating-point type **`double`** occupies two ARM registers if software floating-point is used.

The **`register`** keyword is regarded by the compiler as a suggestion only. Other variables, not declared with the **`register`** keyword, can be kept in registers and register variables can be kept in memory. Using `register` might increase code size because the compiler is restricted in its use of registers for optimization.

3.2.10 Structures, unions, enumerations, and bitfields

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

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

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

These implementation details are documented in the relevant sections of *Standard C implementation definition* on page 3-11 and in the following sections:

Unions Refer to *Unions* for details.

Padding and alignment of structure members
Refer to *Packed structures* on page 3-22 for details.

Enumerations Refer to *Enumerations* for details.

Bitfields Refer to *Bitfields* on page 3-23 for details.

Unions

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

Enumerations

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

- **`unsigned char`**
- **`signed char`**
- **`unsigned short`**
- **`signed short`**
- **`unsigned int`** (C++ always, C except when `-strict`)
- **`signed int`**.

Implementing **`enum`** in this way can reduce data size. The command-line option `-fy` sets the underlying type of **`enum`** to **`signed int`**. Refer to *About the C and C++ compilers on page 2-2* for more information on the `-fy` option.

Unless the `-strict` option is used, **`enum`** declarations may have a comma at the end as in:

```
enum { x = 1, };
```

Structures

The following points apply to:

- all C structures
- all C++ structures and classes not using virtual functions or base classes.

Structure Alignment

The alignment of a non-packed structure is the maximum alignment required by any of its fields.

Field alignment

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

- A field with a **`char`** type is aligned to the next available byte.
- A field with a **`short`** type is aligned to the next even-addressed byte.
- Bitfield alignment depends on how the bitfield is declared. Refer to *Bitfields in packed structures* on page 3-25 for more information.
- All other types are aligned on word boundaries.

Structures can contain padding to ensure that fields are correctly aligned and that the structure itself is correctly aligned. Figure 3-1 on page 3-22 shows an example of a conventional, non-packed structure. Bytes 1, 2, and 3 are padded to ensure correct field alignment. Bytes 10 and 11 are padded to ensure correct structure alignment.

The compiler pads structures in one of two ways, according to how the structure is defined:

- Structures that are defined as **`static`** or **`extern`** are padded with zeros.
- Structures on the stack or heap, such as those defined with `malloc()` or **`auto`**, are padded whatever was previously stored in memory. You cannot use `memcmp()` to compare padded structures defined in this way (see Figure 3-1 on page 3-22).
- Structures with empty initializers are allowed in C++ and only warned about in C (if C and `-strict` an error is generated):

  ```
  struct { int x; } X = { };
  ```

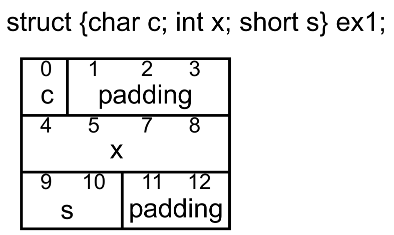


Figure 3-1 Conventional structure example

Packed structures

A packed structure is one where the alignment of the structure, and of the fields within it, is always 1. Floating-point types cannot be fields of packed structures.

Packed structures are defined with the `__packed` qualifier (see *Structures, unions, enumerations, and bitfields* on page 3-20). There is no command-line option to change the default packing of structures.

Bitfields

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

little-endian	Lowest addressed means least significant.
big-endian	Lowest addressed means most significant.

A bitfield container can be any of the integral types.

Note

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

A plain bitfield, declared without either **`signed`** or **`unsigned`** qualifiers, is treated as **`unsigned`**. For example, `int x:10` allocates an unsigned integer of 10 bits.

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

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

The first declaration creates an integer container and allocates 10 bits to `x`. At the second declaration, the compiler finds the existing integer container with a sufficient number of unallocated bits, and allocates `y` in the same container as `x`.

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

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

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

Bitfield containers can *overlap* each other, for example:

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

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

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

If `y` is declared `char y:8`, the compiler pads the second byte and allocates a new **`char`** container to the third byte, because the bitfield cannot overflow its container (see Figure 3-2).

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

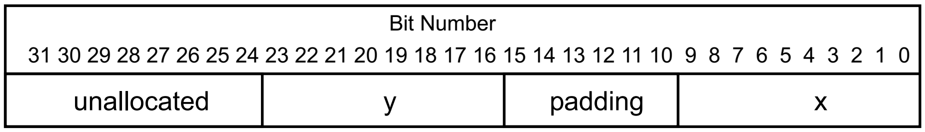


Figure 3-2 Bitfield allocation 1

——— **Note** ———

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

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

The compiler allocates an **`int`** container starting at the same location as the `int x:10` container and allocates a byte-aligned **`char`** and 5-bit bitfield (Figure 3-3).

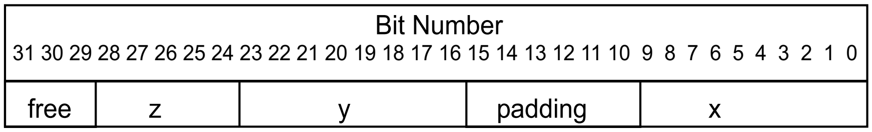


Figure 3-3 Bitfield allocation 2

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

Bitfields in packed structures

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

3.2.11 Qualifiers

This section describes the implementation of various standard C and ARM-specific type qualifiers. These type qualifiers can be used to instruct the compiler to treat the qualified type in a special way. Standard qualifiers that do not have ARM-specific behavior or restrictions are not documented.

__packed

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

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

The `__packed` qualifier cannot be used on:

- floating-point types
- structures or unions with floating-point fields
- structures that were previously declared without `__packed`.

Note

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

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

The `__packed` qualifier applies to all members of a structure or union when it is declared using `__packed`. There is no padding between members, or at the end of the structure. All substructures of a packed structure must be declared using `__packed`. Integral subfields of an unpacked structure can be packed individually.

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

The effect of casting away `__packed` is undefined. The effect of casting a nonpacked structure to a packed structure is undefined. A pointer to an integral type can be legally cast, explicitly or implicitly, to a pointer to a packed integral type.

A pointer can be packed (see Example 3-1).

Example 3-1

```
__packed int *p
```

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

Note

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

The `__packed` qualifier is useful to map a structure to an external data structure, or for accessing unaligned data, but it is generally not useful to save data size because of the relatively high cost of access. The number of unaligned accesses can be reduced by only packing fields in a structure that requires packing.

When a packed object is accessed using a pointer, the compiler generates code that will work and that is independent of the pointer alignment (see Example 3-2).

Example 3-2

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

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

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

volatile

The standard ANSI qualifier **`volatile`** informs the compiler that the qualified type contains data that can be changed from outside the program. The compiler will not attempt to optimize accesses to **`volatile`** types. For example, volatile structures can be mapped onto memory-mapped registers.

In ARM C and C++, a **`volatile`** object is accessed if any word or byte (or halfword on ARM architectures with halfword support) of the object is read or written. For **`volatile`** objects, reads and writes occur as directly implied by the source code, in the order implied by the source code. The effect of accessing a **`volatile short`** is undefined for ARM architectures that do not support halfwords. Accessing volatile `packed` data is undefined.

__weak

Specifies an **`extern`** object declaration that, if not present, does not cause the linker to fault an unresolved reference. If the reference remains unresolved, its value is assumed to be `NULL`. (See also *Function declaration keywords* on page 3-4.)

3.2.12 Declarators

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

3.2.13 Statements

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

Expression evaluation

The compiler performs the usual arithmetic conversions (promotions) set out in the appropriate C or C++ standard before evaluating an expression.

Note

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

Any aspect of evaluation order not prescribed by the relevant standard, can vary between releases of the ARM compilers.

3.2.14 Preprocessing directives

The ANSI standard C header files are stored within the compiler and can be referred to as described in the standard, for example, `#include <stdio.h>`).

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

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

3.2.15 Library functions

The ANSI C library variants are listed in *About the runtime libraries* on page 4-2.

The precise nature of each C library is unique to the particular implementation. 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 might 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 name, line number
  ```

For implementation details of mathematical functions, locale, signals, and input/output see *About the runtime libraries* on page 4-2.

3.3 Standard C++ implementation definition

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

Note

ARM C++ differs from ISO/IEC because the compliance requirements for Embedded C++ (EC++) differ from the requirements for ISO/IEC C++.

This section does not duplicate information that is part of the standard C implementation. See *Standard C implementation definition* on page 3-11.

When used in ANSI C mode, the ARM C++ compilers are identical to the ARM C compiler. Where there is an implementation feature specific to either C or C++, this is noted in the text. For extension to standard C++, see *C and C++ language extensions* on page 3-33.

3.3.1 EC++ support

ARM C++ supports all features required by the definition of Embedded C++ except for argument-dependent name lookup (Koenig lookup).

3.3.2 Integral conversion

(This section is related to section 4.7 of the ISO/IEC standard.) During integral conversion, if the destination type is signed, the value is unchanged if it can be represented in the destination type and bitfield width. Otherwise, the value is truncated to fit the size of the destination type.

3.3.3 Calling a pure virtual function

If a pure virtual function is called, the message will be written to `stderr` and `abort()` will be called. The message will be written only if the image also uses `fputs()` and `stderr`.

3.3.4 Minor features of language support

Table 3-4 shows the minor features of the language supported by this release of ARM C++.

Table 3-4 Minor feature support for language

Minor feature	Support
atexit	Implemented as defined in *The Annotated C++ Reference*, Addison-Wesley, 1991
Namespaces	No
Runtime type identification (RTTI)	Partial. **Typeid** is supported for static types and expressions with non-polymorphic type. See also the restrictions on new style casts.
New style casts	Partial. ARM C++ supports the syntax of new style casts, but does not enforce the restrictions. New style casts behave in the same manner as old style casts.
Array new/delete	Yes
Nothrow **new**	No (but **new** does not throw)
bool type	Yes
wchar_t type	No
explicit keyword	Yes
Static member constants	Yes
extern inline	Yes
Full linkage specification	Yes
for loop variable scope change	Yes
Covariant return types	Yes (but not for non-leftmost base classes)
Default template arguments	Partial (args not dependent on other template args)
Template instantiation directive	Yes
Template specialization directive	Yes
typename keyword	Yes

Table 3-4 Minor feature support for language (Continued)

Minor feature	Support
Member templates	Yes
Partial specialization for class template	Yes
Partial ordering of function templates	Yes
Universal character names	No
Koenig lookup	No

3.3.5 Major features of language support

Table 3-5 shows the major features of the language supported by this release of ARM C++.

Table 3-5 Major feature support for language

Major feature	ISO/IEC standard section	Support
Core language	1 to 13	Yes
Templates	14	Templates are partially supported
Exceptions	15	No
Libraries	17 to 27	Refer to *Standard C++ library implementation definition* on page 3-32 and to Chapter 4 *The C and C++ Libraries*

3.3.6 Standard C++ library implementation definition

Version 2.01 of the Rogue Wave library provides a subset of the library defined in the standard. There are slight differences from the December 1996 version of the ISO/IEC standard. For details of the implementation definition, see *Standard C++ library implementation definition* on page 4-87.

The library can be used with user-defined functions in order to produce target-dependent applications. See *About the runtime libraries* on page 4-2 for more information.

3.4 C and C++ language extensions

This section describes the language extensions supported by the ARM compilers.

3.4.1 C language extensions

The compilers support the ANSI C language extensions described below and in *C and C++ language extensions*. The extensions are not available if the compiler is restricted to compiling strict ANSI C, for example, by specifying the `-strict` compiler option.

// comments

The character sequence `//` starts a comment. As in C++, the comment is terminated by the next newline character.

Note

Comment removal takes place after line continuation, so:

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

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

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

constant expressions

Extended constant expressions are allowed in initializers:

```
int i;
int j = (int)&i; /* not allowed by ANSI/ISO */
```

3.4.2 C and C++ language extensions

This section describes the extensions to both the ANSI C language, and the ISO/IEC C++ language that are accepted by the compilers. See *C language extensions* for those extensions that apply only to C. None of these extensions are available if the compiler is restricted to compiling strict ANSI C or strict ISO/IEC C++. This will be the case, for example, when the `-strict` compiler option is specified.

Identifiers

The `$` character is a legal character in identifiers.

Void returns and arguments

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

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

long long

ARM C and C++ compilers support 64-bit integer types through the type specifier **`long long`** and **`unsigned long long`**. They behave analogously to **`long`** and **`unsigned long`** with respect to the usual arithmetic conversions. **`long long`** is a synonym for `__int64`.

Integer constants can have:

- an `ll` suffix to force the type of the constant to **`long long`**, if it will fit, or to **`unsigned long long`** if it will not fit
- an `llu` (or `ull`) suffix to force the constant to **`unsigned long long`**

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

Also, a plain integer constant is of type **`unsigned long long`** if its value is large enough. There is a warning message from the compiler indicating the change. For example in strict ANSI C, 2147483648 has type **`unsigned long`**. In ARM C++ it has the type **`long long`**. One consequence of this is the value of an expression such as:

```
2147483648 > –1
```

is 0 in strict C and C++, and 1 in ARM C and C++.

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

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

-

Inline assembler

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

The ARM C++ compilers support the syntax in the ISO/IEC C++ standard, with the restriction that the string-literal must be a single string, for example:

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

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

In addition to the syntax in the ISO/IEC standard, ARM C++ supports the C compiler `__asm` syntax when used with both **`asm`** and `__asm`.

The ARM compilers support the full ARM instruction set, including generic coprocessor instructions, but not `BX` and `BLX`.

The Thumb compilers support the full Thumb instruction set except for `BX` and `BLX`.

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

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

If two instructions are on the same line, you must separate them with a semicolon. If an instruction requires more than one line, line continuation must be specified with the backslash character `\`. C or C++ comments can be used anywhere within an inline assembly language block.

An **`asm`** statement can be used anywhere a C++ statement is expected. The `__asm` keyword is a synonym for **`asm`** and is provided to support compatibility with C.

Refer to Mixed Language Programming in the *ADS Developer Guide* for more information on inline C and C++ assemblers.

Keywords

ARM implements some keyword extensions for functions and variables. See *Function declaration keywords* on page 3-4, *Variable declaration keywords* on page 3-7, and *Qualifiers* on page 3-26.

Hexadecimal floating-point constants

ARM implements an extension to the syntax of numeric constants in C to allow explicit specification of floating-point constants as IEEE bit patterns. The syntax is:

0f_*n*	Interpret an 8-digit hex number *n* as a float
0d_*nn*	Interpret a 16-digit hex number *nn* as a double.

There must be exactly 8 digits for float constants. There must be exactly 16 digits for double constants.

Read/write constants

For C++ only, a new linkage specification for external constants indicates that a constant can be dynamically initialized or have mutable members.

Note

The use of `"C++:read/write"` linkage is only necessary for code compiled `/ropi` or `/rwpi`. If you recompile existing code with either of these options, you will need to change the linkage specification for external constants that are dynamically initialized or have mutable members.

Compiling C++ with either the `/ropi` or `/rwpi` options deviates from the C++ standard. The declarations in Example 3-3 assume that x is in a read-only segment:

Example 3-3

```
extern const T x;
extern "C++" const T x;
extern "C" const T x;
```

Dynamic initialization of `x` (including user-defined constructors) will not be possible for the constants and `T` may not contain mutable members. The new linkage specification in Example 3-4 declares that `x` is in a read/write segment (even if it was initialized with a constant). Dynamic initialization of `x` is allowed and `T` may contain mutable members. The definitions of `x`, `y`, and `z` in another file must have the same linkage specifications.

Example 3-4

```
extern const int z;      /* in read-only segment, cannot  */
                          /* be dynamically initialized    */

extern "C++:read/write" const int y; /* in read/write segment */
                          /* can be dynamically initialized */
extern "C++:read/write" {
  const int i=5;       /* placed in read-only segment, */
                       /* not extern because implicitly static */
  extern const T x=6;     /* placed in read/write segment */
  struct S {
     static const T T x; /* placed in read/write segment */
  };
}
```

Constant objects must not be redeclared with another linkage. The code in Example 3-5 produces a compile error.

Example 3-5

```
extern "C++"  const T x;
extern "C++:read/write"  const T x; /* error */
```

Note

Since C does not have the linkage specifications, it is not possible to use a **`const`** object declared in C++ as `extern "C++:read/write"` from C.

3.5 Predefined macros

Table 3-6 lists the macro names predefined by the ARM C and C++ compilers. Where the value field is empty, the symbol concerned is merely defined, as though by `-D__arm` on the command line.

Table 3-6 Predefined macros

Name	Value	When defined
`__arm`	–	If using armcc, tcc, armcpp, or tcpp.
`__ARMCC_VERSION`	*ver*	For giving the version number of the compiler. The value is the same for the ARM and Thumb compilers. It is a decimal number, whose value can be relied on to increase between releases. The format is *PVtbbb* where *P* is the product (1 for ADS) *V* is the minor version (0 for 1.0) *t* is the patch release (0 for 1.0) *bbb* is the build (103 for example). The example given results in 100103.
`__APCS_INTERWORK`	–	If `-apcs /interwork` in use.
`__APCS_ROPI`	–	If `-apcs /ropi` in use.
`__APCS_RWPI`	–	If `-apcs /rwpi` in use.
`__APCS_SWST`	–	If `-apcs /swst` in use.
`__BIG_ENDIAN`	–	If compiling for a big-endian target.
`__cplusplus`	–	In C++ compiler mode.
`__CC_ARM`	–	Returns compiler name.
`__DATE__`	*Date*	When date of translation of source file is required.
`__embedded_cplusplus`	–	If in EC++ compiler mode.
`__FEATURE_SIGNED_CHAR`	–	Set by `-zc` (used by `CHAR_MIN` and `CHAR_MAX`).
`__FILE__`	*name*	The presumed full pathname of the current source file.
`__func__`	*name*	The name of the current function.
`__LINE__`	*num*	When line number of the current source file is required.

-

Table 3-6 Predefined macros (Continued)

Name	Value	When defined
`__MODULE__`	*mod*	Contains the filename part of the value of `__FILE__`.
`__OPTIMISE_SPACE`	–	If `-Ospace` in use.
`__OPTIMISE_TIME`	–	If `-Otime` in use.
`__prettyfunc__`	*name*	The unmangled name of the current function.
`__sizeof_int`	4	For `sizeof(int)`, but available in preprocessor expressions.
`__sizeof_long`	4	For `sizeof(long)`, but available in preprocessor expressions.
`__sizeof_ptr`	4	For `sizeof(void *)`, but available in preprocessor expressions.
`__SOFTFP__`	–	If compiling to use the software floating-point library (`-apcs /softfp`).
`__STDC__`	–	In all compiler modes.
`__STDC_VERSION__`	–	standard version information.
`__STRICT_ANSI__`	–	Set by `-strict`.
`__TARGET_ARCH_`*xx*	–	*xx* represents the target architecture and its value depends on the target architecture. For example, if the compiler options `-cpu 4T` or `-cpu ARM7TDMI` are specified then `__TARGET_ARCH_4T` is defined, and no other symbol starting with `_TARGET_ARCH_` is defined.
`__TARGET_CPU_`*xx*	–	*xx* represents the target cpu. The value of *xx* is derived from the `-cpu` compiler option, or the default if none is specified. For example, if the compiler option `-cpu ARM7TM` is specified then `_TARGET_CPU_ARM7TM` is defined and no other symbol starting with `_TARGET_CPU_` is defined. If the target architecture is specified, then `_TARGET_CPU_generic` is defined. If the processor name contains hyphen (-) characters, these are mapped to an underscore (_). For example, `-cpu SA-110` is mapped to `__TARGET_CPU_SA_110`.

Table 3-6 Predefined macros (Continued)

Name	Value	When defined
`__TARGET_FEATURE_DSPMUL`	–	If the DSP-enhanced multiplier is available.
`__TARGET_FEATURE_HALFWORD`	–	If the target architecture supports halfword and signed byte access instructions.
`__TARGET_FEATURE_MULTIPLY`	–	If the target architecture supports the long multiply instructions `MULL` and `MULAL`.
`__TARGET_FEATURE_THUMB`	–	If the target architecture is Thumb-aware.
`__TARGET_FPU_`*`xx`*	–	Identifies the FPU option as one of `__TARGET_FPU_VFP`, `__TARGET_FPU_FPA`, `__TARGET_FPU_SOFTVFP` `__TARGET_FPU_SOFTFPA`, or `__TARGET_FPU_NONE`
`__thumb`	–	If using tcc or tcpp.
`__TIME__`	*Time*	When time of translation of the source file is required.

-

3.6 Implementation limits

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

3.6.1 C++ ISO/IEC Standard Limits

The ISO/IEC C++ standard recommends minimum limits that a conforming compiler should accept. You should be aware of these when porting applications between compilers. A summary is given in Table 3-7. A limit of memory indicates that no limit is imposed by the ARM compilers, other than that imposed by the available memory.

Table 3-7 Implementation limits

Description	Recommended	ARM
Nesting levels of compound statements, iteration control structures, and selection control structures.	256	memory
Nesting levels of conditional inclusion.	256	memory
Pointer, array, and function declarators (in any combination) modifying an arithmetic, structure, union, or incomplete type in a declaration.	256	memory
Nesting levels of parenthesized expressions within a full expression.	256	memory
Number of initial characters in an internal identifier or macro name.	1024	1024
Number of initial characters in an external identifier.	1024	1024
External identifiers in one translation unit.	65536	memory
Identifiers with block scope declared in one block.	1024	memory
Macro identifiers simultaneously defined in one translation unit.	65536	memory
Parameters in one function declaration. Overload resolution is sensitive to the first 32 arguments only.	256	memory
Arguments in one function call. Overload resolution is sensitive to the first 32 arguments only.	256	memory
Parameters in one macro definition.	256	memory
Arguments in one macro invocation.	256	memory
Characters in one logical source line.	65536	memory
Characters in a character string literal or wide string literal after concatenation.	65536	memory

Table 3-7 Implementation limits (Continued)

Description	Recommended	ARM
Size of a C++ object.	262144	8388607
Nesting levels of #include file.	256	memory
Case labels for a switch statement, excluding those for any nested switch statements.	16384	memory
Data members in a single class, structure, or union.	16384	memory
Enumeration constants in a single enumeration.	4096	memory
Levels of nested class, structure, or union definitions in a single struct-declaration-list.	256	memory
Functions registered by `atexit()`.	32	33
Direct and indirect base classes	16384	memory
Direct base classes for a single class	1024	memory
Members declared in a single class	4096	memory
Final overriding virtual functions in a class, accessible or not	16384	memory
Direct and indirect virtual bases of a class	1024	memory
Static members of a class	1024	memory
Friend declarations in a class	4096	memory
Access control declarations in a class	4096	memory
Member initializers in a constructor definition	6144	memory
Scope qualifications of one identifier	256	memory
Nested external specifications	1024	memory
Template arguments in a template declaration	1024	memory
Recursively nested template instantiations	17	memory
Handlers per try block	256	memory
Throw specifications on a single function declaration	256	memory

3.6.2 Internal limits

In addition to the limits described in Table 3-7 on page 3-41, the compiler has internal limits as listed in Table 3-8.

Table 3-8 Internal limits

Description	**ARM**
Maximum number of relocatable references in a single translation unit.	65536
Maximum number of virtual registers.	65536
Maximum number of overload arguments.	32
Number of characters in a mangled name before it will be truncated.	4096
Number of bits in the smallest object that is not a bit field (`CHAR_BIT`).	8
Maximum number of bytes in a multibyte character, for any supported locale (`MB_LEN_MAX`).	1

3.7 Limits for integral numbers

Table 3-9 gives the ranges for integral numbers in ARM C and C++. The third column of the table gives the numerical value of the range endpoint. The fourth column gives the bit pattern (in hexadecimal) that would be interpreted as this value by the ARM compilers.

When entering a constant, choose the size and sign with care. Constants are interpreted differently in decimal and hexadecimal/octal. See the appropriate C or C++ standard, or any of the recommended C and C++ textbooks for more details (refer to *Further reading* on page Preface-iii).

Table 3-9 Integer ranges

Constant	Meaning	Endpoint	Hex value
`CHAR_MAX`	Maximum value of **`char`**	255	`0xff`
`CHAR_MIN`	Minimum value of **`char`**	0	`0x00`
`SCHAR_MAX`	Maximum value of **`signed char`**	127	`0x7f`
`SCHAR_MIN`	Minimum value of **`signed char`**	–128	`0x80`
`UCHAR_MAX`	Maximum value of **`unsigned char`**	255	`0xff`
`SHRT_MAX`	Maximum value of **`short`**	32767	`0x7fff`
`SHRT_MIN`	Minimum value of **`short`**	–32768	`0x8000`
`USHRT_MAX`	Maximum value of **`unsigned short`**	65535	`0xffff`
`INT_MAX`	Maximum value of **`int`**	2147483647	`0x7fffffff`
`INT_MIN`	Minimum value of **`int`**	–2147483648	`0x80000000`
`LONG_MAX`	Maximum value of **`long`**	2147483647	`0x7fffffff`
`LONG_MIN`	Minimum value of **`long`**	–2147483648	`0x80000000`
`ULONG_MAX`	Maximum value of **`unsigned long`**	4294967295	`0xffffffff`
`LONG_LONG_MAX`	Maximum value of **`long long`**	9.2E+18	`0x7fffffff ffffffff`
`LONG_LONG_MIN`	Minimum value of **`long long`**	–9.2E+18	`0x80000000 00000000`
`ULONG_LONG_MAX`	Maximum value of **`unsigned long long`**	1.8E+19	`0xffffffff ffffffff`

3.8 Limits for floating-point numbers

Table 3-10 and Table 3-11 give the characteristics, ranges, and limits for floating-point numbers in ARM and Thumb compilers.

Note

When a floating-point number is converted to a shorter floating-point number, it is rounded to the nearest representable number.

The properties of floating-point arithmetic accord with IEEE 754.

Table 3-10 Floating-point limits

Constant	Meaning	Value
`FLT_MAX`	Maximum value of **`float`**	3.40282347e+38F
`FLT_MIN`	Minimum value of **`float`**	1.17549435e–38F
`DBL_MAX`	Maximum value of **`double`**	1.79769313486231571e+308
`DBL_MIN`	Minimum value of **`double`**	2.22507385850720138e–308
`LDBL_MAX`	Maximum value of **`long double`**	1.79769313486231571e+308
`LDBL_MIN`	Minimum value of **`long double`**	2.22507385850720138e–308
`FLT_MAX_EXP`	Maximum value of base 2 exponent for type **`float`**	128
`FLT_MIN_EXP`	Minimum value of base 2 exponent for type **`float`**	–125
`DBL_MAX_EXP`	Maximum value of base 2 exponent for type **`double`**	1024
`DBL_MIN_EXP`	Minimum value of base 2 exponent for type **`double`**	–1021
`LDBL_MAX_EXP`	Maximum value of base 2 exponent for type **`long double`**	1024
`LDBL_MIN_EXP`	Minimum value of base 2 exponent for type **`long double`**	–1021
`FLT_MAX_10_EXP`	Maximum value of base 10 exponent for type **`float`**	38
`FLT_MIN_10_EXP`	Minimum value of base 10 exponent for type **`float`**	–37
`DBL_MAX_10_EXP`	Maximum value of base 10 exponent for type **`double`**	308

Table 3-10 Floating-point limits (Continued)

Constant	Meaning	Value
`DBL_MIN_10_EXP`	Minimum value of base 10 exponent for type **`double`**	–307
`LDBL_MAX_10_EXP`	Maximum value of base 10 exponent for type **`long double`**	308
`LDBL_MIN_10_EXP`	Minimum value of base 10 exponent for type **`long double`**	–307

Table 3-11 Other floating-point characteristics

Constant	Meaning	Value
`FLT_RADIX`	Base (radix) of the ARM floating-point number representation	2
`FLT_ROUNDS`	Rounding mode for floating-point numbers	(nearest) 1
`FLT_DIG`	Decimal digits of precision for **`float`**	6
`DBL_DIG`	Decimal digits of precision for **`double`**	15
`LDBL_DIG`	Decimal digits of precision for **`long double`**	15
`FLT_MANT_DIG`	Binary digits of precision for type **`float`**	24
`DBL_MANT_DIG`	Binary digits of precision for type **`double`**	53
`LDBL_MANT_DIG`	Binary digits of precision for type **`long double`**	53
`FLT_EPSILON`	Smallest positive value of x that 1.0 + x != 1.0 for type **`float`**	1.19209290e–7F
`DBL_EPSILON`	Smallest positive value of x that 1.0 + x != 1.0 for type **`double`**	2.2204460492503131e–16
`LDBL_EPSILON`	Smallest positive value of x that 1.0 + x != 1.0 for type **`long double`**	2.2204460492503131e–16L

Chapter 4
The C and C++ Libraries

This chapter describes the ARM C and C++ libraries. The libraries support programs written in C or C++. This chapter contains the following sections:

4.1 About the runtime libraries

The following runtime libraries are provided to support compiled C and C++:

ANSI C The C libraries consist of:

- The functions defined by the ISO C library standard.
- Target-dependent functions used to implement the C library functions in the semihosted execution environment. You can redefine these functions in your own application.
- Helper functions used by the C and C++ compilers.

C++ The C++ libraries contain the functions defined by the ISO C++ library standard. The C++ library depends on the C library for target-specific support and there are no target dependencies in the C++ library. This library consists of:

- the Rogue Wave Standard C++ Library version 2.0.1
- helper functions for the C++ compiler
- additional C++ functions not supported by the Rogue Wave library.

For a detailed description of how the libraries comply with the ISO standard, see *ISO implementation definition* on page 4-81.

As supplied, the ANSI C libraries use the standard ARM semihosted environment to provide facilities such as file input/output. This environment is supported by the ARMulator, Angel, Multi-ICE, and EmbeddedICE. You can use the ARM development tools in ADS to develop applications, and then immediately run and debug the applications under the ARMulator or on a development board. See the description of semihosting in the *ADS Debug Target Guide* for more information on the debug environment.

You can re-implement any of the target-dependent functions of the C library as part of your application. This lets you tailor the C library, and therefore the C++ library, to your own execution environment.

You can also tailor many of the target-independent functions to your own application-specific requirements, for example:

- the malloc family
- the ctype family
- all of the locale-specific functions.

Many of the C library functions are independent of any other function and contain no target dependencies. You can easily exploit these functions from assembly language.

4.1.1 Build options and library variants

When you build your application, you must make certain fundamental choices. For example:

Byte order Big-endian or little-endian.

Floating-point support
FPA, VFP, software, or none.

Stack limit Checked or unchecked.

Position-independence
Data can be read/write position-independent or not position-independent, code can be read-only position-independent or not position-independent.

When you link your assembly language, C, or C++ code, the linker selects appropriate C and C++ library variants compatible with the build options you specified. There is a variant of the ANSI C library for each combination of major build options. Build options are described in more detail in:

- the ATPCS chapter in the *ADS Developer Guide*
- *Selecting library variants* on page 6-27 for the linker
- *Procedure Call Standard options* on page 2-10 for the compiler
- *Command syntax* on page 5-4 for the assembler.

4.1.2 Library directory structure

The libraries are installed in two subdirectories within `install_directory\lib`:

`armlib`	This subdirectory contains the variants of the ARM C library, the floating-point arithmetic library, and the math library. The accompanying header files are in `install_directory\include`.
`cpplib`	This subdirectory contains the variants of the Rogue Wave C++ library and supporting C++ functions. The Rogue Wave and supporting C++ functions are collectively referred to as the ARM C++ Libraries. The accompanying header files are installed in `install_directory\include`.

The environment variable ARMLIB should be set to point to the `lib` directory. Alternatively use the `-libpath` argument to the linker to identify the directory holding the library subdirectories.

There is no need to identify `armlib` and `cpplib` separately. The linker finds them for you from the location of `lib`.

Note

- The ARM C libraries are supplied in binary form only.
- The ARM libraries should not be modified. If you want to create a new implementation of a library function, place the new function in an object file, or your own library, and include it when you link the application. Your version of the function will be used instead of the standard library version.
- Normally, only a few functions in the ANSI C library require re-implementation in order to create a target-dependent application.
- The source for the Rogue Wave Standard C++ Library is not freely distributable. It can be obtained from Rogue Wave Software Inc., or through ARM Ltd, for an additional licence fee. See the Rogue Wave online documentation in `install_directory\Html` for more about the C++ library.

4.1.3 Reentrancy and static data

Libraries that make use of static data are supplied in two variants:

- Static data addressed in a position-dependent fashion. Code from these variants is single threaded.
- Static data addressed in a position-independent fashion using offsets from the static base register sb (r9). Code from these variants can be multiply-threaded and is reentrant.

The following points describe how static data is used by the libraries:

- Floating-point arithmetic libraries do not use static data and are always reentrant.
- All statically-initialized data in the C libraries is read-only.
- All writable static data is uninitialized.
- Most C library functions use no writable static data and are reentrant whether built with base build options (`-apcs /norwpi`) or reentrant (`-apcs /rwpi`) build options.
- A few functions have static data in their definitions, for example `strtok()`, `localtime()`, `gmtime()`, `rand()`, and `srand()`. You should avoid using these functions in a reentrant application unless you build it `/rwpi`.

4.2 Building an application with the C library

This section covers creating an application that links with functions from the C or C++ libraries. Functions in the C library are responsible for:

- Creating an environment in which a C or C++ program can execute. This includes
  - creating a stack
  - creating, if required, a heap
  - initializing the parts of the library the program uses.
- Starting execution by calling `main()`.
- Supporting use of ISO-defined functions by the program.
- Catching run-time errors and signals and, if required, terminating execution on error or program exit.

There are three major ways to use the libraries with an application:

- Build a semihosted application which can be debugged in a semihosted environment such as with ARMulator or Angel. See *Building an application for a semihosted environment* on page 4-6.
- Build a non-hosted application which can, for example, be embedded into ROM. See *Building an application for a non-semihosted environment* on page 4-8.
- Build an application which does not use `main()` and does not initialize the library. This application will have, unless you re-implement some functions, restricted library functionality. See *Building an application without the C library* on page 4-13.

4.2.1 Building an application for a semihosted environment

If you are developing an application that will run in a semihosted environment for debugging, you must have an execution environment that supports the ARM and Thumb semihosting SWIs and has sufficient memory.

The execution environment can be provided by either:

- using the standard semihosting functionality that is present by default in, for example, ARMulator, Angel, and Multi-ICE
- implementing your own SWI handler for the semihosting SWI (see *ADS Debug Target Guide*).

A list of functions that require semihosting is given in *Overview of semihosting dependencies* on page 4-9.

You do not need to write any new functions or include files if you are using the default semihosting functionality of the library.

Using ARMulator

The ARM instruction set simulator (*ARMulator*) supports the semihosting SWI and has adequate memory maps for using the library. The ARMulator will of course use memory in the host machine and this will normally be adequate for your application.

Using Angel

ARM boards running the Angel debug monitor support the semihosting SWI and have adequate memory maps for using the library. Your application might, however, require more memory than is available on the development board and the memory map assumed by the library might require tailoring to match the hardware being debugged

You can change the definition of the Angel environment. See the *ADS Debug Target Guide* for more information on semihosting and the Angel environment.

Using Multi-ICE and EmbeddedICE

The ARM debug agents support the semihosting SWI, but the memory map assumed by the library might require tailoring to match the hardware being debugged. However, it is easy to tailor the memory map assumed by the C library. See *Tailoring the run-time memory model* on page 4-60.

Using re-implemented functions in a semihosted environment

You can also mix the semihosting functionality with new input/output functions. For example, you could implement `putc()` to output directly to hardware, for example to a UART, in addition to the semihosted implementation. See *Building an application for a non-semihosted environment* for information on how to re-implement individual functions.

Converting a semihosted application to a standalone application

After an application has been developed in a semihosted debugging environment, you can move the application to a non-hosted environment by one of the following methods:

- Removing all calls to semihosted functions. See *Avoiding the semihosting SWI* on page 4-10.
- Re-implementing the semihosted functions. See *Building an application for a non-semihosted environment* on page 4-8. You do not have to re-implement all semihosted functions. You must, however, re-implement the functions that you are using in your application.
- Implementing a SWI handler that handles the semihosting SWIs.

Implementing your own semihosting SWI support

It is possible to implement your own semihosting SWI support. The interface is simple and requires a handler for just two SWI numbers. `0x123456` is used in ARM state and `0xab` is used in Thumb state. See the semihosting SWI definitions in *ADS Debug Target Guide* and the include file `rt_sys.h`

4.2.2 Building an application for a non-semihosted environment

If you do not want to use any semihosting functionality, you must ensure that either no calls are made to any function that uses semihosting or that such functions are replaced by your own non-semihosted functions.

To build an application that does not use semihosting functionality:

1. Create the source files to implement the target-dependent features.
2. Add the `__use_no_semihosting_swi()` guard to the source. See *Avoiding the semihosting SWI* on page 4-10.
3. Link the new objects with your application.
4. Use the new configuration when creating the target-dependent application.

You must re-implement functions that the C library uses to insulate itself from target dependencies. For example, if you use `printf()` you will need to re-implement `fputc()`. If you do not use the higher-level input/output functions like `printf()`, you do not need to re-implement the lower-level functions like `fputc()`.

If you are building an application for a different execution environment, you can re-implement the target dependent functions (functions that use the semihosting SWI or that depend on the target memory map). There are no target-dependent functions in the C++ library.

The functions that you might have to re-implement are described in:

- *Tailoring static data access* on page 4-23
- *Tailoring locale and CTYPE* on page 4-24
- *Tailoring error signalling, error handling, and program exit* on page 4-47
- *Tailoring the run-time memory model* on page 4-60
- *Tailoring the input/output functions* on page 4-67
- *Tailoring other C library functions* on page 4-76

Examples of embedded applications that do not use a hosted environment are included in `*install_directory*\examples\ROM`.

See the *ADS Developer Guide* for examples of creating applications for embedding into ROM.

-

Overview of semihosting dependencies

The functions shown in Table 4-1 depend directly upon semihosting SWIs:

Table 4-1 Direct dependencies

Function	Description
`__user_initial_stackheap()`	*Tailoring the run-time memory model* on page 4-60
`_sys_exit()`	*Tailoring error signalling, error handling, and program exit* on page 4-47
`_ttywrch()`	
`_sys_command_string()`	*Tailoring the input/output functions* on page 4-67
`_sys_close(),_sys_ensure(), _sys_iserror(), _sys_istty(), _sys_flen(), _sys_open(), _sys_read(), _sys_seek(), _sys_write()`	
`_sys_tmpnam()`	
`time()`	*Tailoring other C library functions* on page 4-76
`remove()`	
`rename()`	
`system()`	
`clock(), _clock_init()`	

The functions listed in Table 4-2 depend indirectly upon one or more of the functions listed in Table 4-1:

Table 4-2 Indirect dependencies

Function	Where used
`__raise()`	Catch, handle, or diagnose C library exceptions, without C signal support. See *Tailoring error signalling, error handling, and program exit* on page 4-47
`__default_signal_handler()`	Catch, handle, or diagnose C library exceptions, with C signal support. See *Tailoring error signalling, error handling, and program exit* on page 4-47
`__Heap_Initialize()`	Choosing or redefining memory allocation. See *Tailoring storage management* on page 4-52
`ferror(), fputc(), __stdout`	Retargeting the printf family. See *Tailoring the input/output functions* on page 4-67
`__backspace(), fgetc(), __stdin`	Retargeting the scanf family. See *Tailoring the input/output functions* on page 4-67
`fwrite(), fputs(), puts(), fread(), fgets(), gets(), ferror()`	Retargeting the stream output family. See *Tailoring the input/output functions* on page 4-67

Avoiding the semihosting SWI

If you write an application in C, you must link it with the C library even if it makes no direct use of C library functions. The C library contains compiler helper functions and initialization code. Some C library functions use the semihosting SWI. To avoid using the semihosting SWI, do either of the following:

- re-implement the functions in your own application
- write the application so that it does not call any semihosted function.

To guarantee that no functions using the semihosting SWI are included in your application, make the following call at any point in your C program:

```
__use_no_semihosting_swi();
```

Its declaration, given in `<stdio.h>`, is:

```
extern void __use_no_semihosting_swi(void);
```

Place the following import in your assembler program:

```
IMPORT __use_no_semihosting_swi
```

The function has no effect except to cause a link-time error if a function that uses the semihosting SWI is included from the library. The linker error message is:

```
duplicate definition of __semihosting_swi_guard
```

Use the linker symbol table and cross reference listings to identify functions you have called that directly, or indirectly, use semihosting. This information can be viewed by using the linker options `-map`, `-xref`, and `-v`. Remove, or re-implement semihosted functions, and rebuild the application.

API definitions

In addition to the semihosted functions listed in Table 4-1 and Table 4-2, the functions and files listed in Table 4-3 might be useful when building for a different environment.

Table 4-3 Published API definitions

File or function	Description
`__main()` and `__rt_entry()`	Initializes the runtime environment and executes the user application.
`__rt_lib_init()`, `__rt_exit()`, and `__rt_lib_shutdown()`	Initializes or finalizes the runtime library.
`locale()` and `CTYPE`	Defines the character properties for the local alphabet. See *Tailoring locale and CTYPE* on page 4-24
`rt_sys.h`	A C header file describing all the functions whose default (semihosted) implementations use the semihosting SWI. .
`rt_heap.h`	A C header file describing the storage management abstract data type.
`rt_locale.h`	A C header file describing the five locale category *filing systems*, and defining some macros that are useful for describing the contents of locale categories.
`rt_misc.h`	A C header file describing miscellaneous unrelated public interfaces to the C library.
`rt_memory.s`	An empty, but commented, prototype implementation of the memory model manager.

If you are re-implementing a function that exists in the standard ARM library, the linker will use an object or library from your project rather than the standard ARM library. A library you add to a project does not have to follow the ARM naming convention for libraries.

Note

You must not use the same name for one of your libraries that ARM also uses for the supplied libraries.

4.3 Building an application without the C library

Creating an application that has a `main()` function causes the C library initialization functions to be included.

If your application does not have a `main()` function, the C library will not be initialized and the following features will not be available in your application:

- software stack checking
- low-level stdio
- signal-handling functions, `signal()` and `raise()` in `signal.h`
- `atexit()`
- `alloca()`.

This section refers to creating applications without the library as *bare machine C*. These applications will not automatically use the full C run-time environment provided by the C library. Even though you are creating an application without the library, some helper functions from the library must be included. There are also many library functions that can be made available with only minor re-implementations.

4.3.1 Integer and FP helper functions

There are several compiler helper functions that are used by the compiler to handle operations that do not have a short machine code equivalent. For example, integer divide uses a helper function because there is not a divide instruction in the ARM and Thumb instruction set.

Integer divide and all the floating-point functions require `__rt_raise()` to handle math errors. Re-implementing `__rt_raise()` enables all the math helper functions.

4.3.2 Bare machine integer C

If you are writing a program in C that is to run without any environment initialization you must:

- Implement `__rt_raise()` yourself, since this error-handling function can be called from numerous places within the compiled code.
- Not define `main()` in order to avoid linking in the library initialization code.
- Not use software stack checking in the build options.
- Write an assembly language veneer that establishes the register state needed to run C. This veneer should branch to the entry function in your application.
- Ensure that your initialization veneer is executed by, for example, placing it in your reset handler.
- Build your application using `-fpu none` and link it normally. The linker will use the appropriate C library variant to find any needed compiler helper functions.

Many library facilities require `__user_libspace()` for static data. Even without the initialization code activated by having a `main()` function, `__user_libspace()` will be created automatically and use 64 bytes in the ZI segment.

4.3.3 Bare machine C with floating-point

If you want to use floating-point processing in your application you must:

- perform the steps necessary for integer C as described above in *Bare machine integer C*
- use the appropriate FPU option when you build your application
- call `_fp_init()` to initialize the floating-point status register before performing any floating-point operations.

If you are using software floating-point, you must also:

- define the function `__rt_fp_status_addr()` to return the address of a writable data word that will be used instead of the floating-point status register.

4.3.4 Exploiting the C library

If you create an application that includes a `main()` function, the linker will automatically include the initialization code necessary for the execution environment. See *Building an application with the C library* on page 4-6 for instructions. There are situations though where this is not desirable or possible.

You can create an application that consists of customized startup code and still use many of the library functions. You must either:

- avoid functions that require initialization
- provide the initialization and low-level support functions.

Program design

The functions you must re-implement depend on how much of the library functionality you require as follows:

- If you want only the compiler support functions for division, structure copy, and FP arithmetic, you must provide `__rt_raise()`. This also allows very simple library functions such as those in `errno.h`, `setjmp.h`, most of `string.h` to work.
- If you call `setlocale()` explicitly, locale-dependent functions will start to work. This allows you to use the `atoi` family, `sprintf()`, `sscanf()`, and the functions in `ctype.h`
- Programs that use floating-point must call `_fp_init()`. If you select software floating-point, the program must also provide `__rt_fp_status_addr()`.
- Implementing high-level input/output support is necessary for functions that use `fprintf()` or `fputs()`. The high-level output functions depend on `fputc()` and `ferror()`. The high-level input functions depend on `fgetc()` and `__backspace()`.
- Implementing the above functions and the heap allows use of almost the entire library.

Using low-level functions

If you are using the libraries in an application that does not have a `main()` function, you must re-implement some functions in the library. See *The standalone C library functions* on page 4-16 for a detailed list of functions are not available, functions that are available without modification, and functions that are available after other lower-level functions are re-implemented.

`__rt_raise()` is essential. It is required by all FP functions, by integer division so that divide-by-zero can be reported, and by some other library routines. You probably cannot write a non-trivial program without doing something that requires it.

Note

If `rand()` is called, `srand()` *must* be called first. This is done automatically during library initialization but not when you avoid the library initialization.

Using high-level functions

High-level I/O functions, `fprintf()` for example, can be used if the low-level functions, `fputc()` for example, are re-implemented. Most of the formatted output functions will also require a call to `setlocale()`. See *Tailoring the input/output functions* on page 4-67 for instructions.

Anything that uses locale should not be called before first calling `setlocale()` to initialize it, for example call `setlocale(LC_ALL, "C")`. Locale-using functions are described in *The standalone C library functions* on page 4-16. These include the functions in `ctype.h` and `locale.h`, the `printf` family, the `scanf` family, `ato*`, `strto*`, `strcoll/strxfrm`, and much of `time.h`.

Using malloc()

If heap support is required for bare machine C, `_init_alloc()` must be called first to supply initial heap bounds, and `__rt_heap_extend()` *must* be provided even if it just returns failure. Prototypes for both functions are in `rt_heap.h`.

4.3.5 The standalone C library functions

The following sections list the include files and the functions in them that are available with an uninitialized library. Some otherwise unavailable functions can be used if the library functions they depend on are re-implemented.

alloca.h

Functions listed in this file are not available without library initialization. See *Building an application with the C library* on page 4-6 for instructions.

assert.h

Functions listed in this file require high-level stdio, `__rt_raise()`, and `_sys_exit()`. See *Tailoring error signalling, error handling, and program exit* on page 4-47 for instructions.

ctype.h

Functions listed in this file require the `locale` functions.

errno.h

Functions in this file work without the need for any library initialization or function re-implementation.

fenv.h

Functions in this file work without the need for any library initialization and only require the re-implementation of `__rt_raise()`.

float.h

This file does not contain any code. The definitions in the file do not require library initialization or function re-implementation.

inttypes.h

Functions listed in this file require the `locale` functions.

limits.h

Functions in this file work without the need for any library initialization or function re-implementation.

locale.h

Call `setlocale()` before calling any function that uses `locale` functions. For example call:

```
setlocale(LC_ALL, "C")
```

See the contents of `locale.h` for details of the following functions and data structures:

`setlocale()`	Selects the appropriate locale as specified by the category and locale arguments.
`lconv`	Is the structure used by `locale` functions for formatting numeric quantities according to the rules of the current locale.
`localeconv()`	Creates an `lconv` structure and returns a pointer to it.
`_get_lconv()`	Fills the `lconv` structure pointed to by the parameter. This ANSI extension removes the need for static data within the library.

`locale.h` also contains constant declarations used with locale functions. See *Tailoring locale and CTYPE* on page 4-24 for more information.

math.h

Functions in this file work without the need for any library initialization and only require the re-implementation of `__rt_raise()`. You must of course call `_fp_init()` in order to use floating-point functions.

setjmp.h

Functions in this file work without the need for any library initialization or function re-implementation.

signal.h

Functions listed in this file are not available without library initialization. See *Building an application with the C library* on page 4-6 for instructions on building an application that uses library initialization.

`__rt_raise()` can be re-implemented for error and exit handling. See *Tailoring error signalling, error handling, and program exit* on page 4-47 for instructions.

stdarg.h

Functions in this file work without the need for any library initialization or function re-implementation.

stddef.h

This file does not contain any code. The definitions in the file do not require library initialization or function re-implementation.

stdint.h

This file does not contain any code. The definitions in the file do not require library initialization or function re-implementation.

stdio.h

The following dependencies or limitations apply to these files:

- The high-level functions such as `printf()`, `scanf()`, `puts()`, `fgets()`, `fread()`, `fwrite()`, `perror()` and so on require high-level stdio. See *Tailoring the input/output functions* on page 4-67 for instructions.
- The `printf()` and `scanf()` family of functions require `locale`.
- The `remove()` and `rename()` functions are system-specific and probably not usable in your application.

stdlib.h

Most functions in this file work without the need for any library initialization or function re-implementation. The following are not available or require implementation of a support function:

`ato*()`	requires `locale`
`strto*()`	requires `locale`
`malloc()`	`malloc()`, `calloc()`, `realloc()`, and `free()` require heap functions
`atexit()`	is not available.

string.h

Functions in this file work without the need for any library initialization with the exception of `strcoll()` and `strxfrm()` that require `locale`.

time.h

`mktime()` and `localtime()` can be used immediately.

`time()` and `clock()` are system-specific and probably not usable unless re-implemented.

`asctime()`, `ctime()`, and `strftime()` require `locale`.

4.4 Tailoring the C library to a new execution environment

This section describes how to re-implement functions to produce an application for a different execution environment, for example embedded in ROM or used with an RTOS.

Symbols that have a single or double underscore, _ or __, name functions that are used as part of the low-level implementation. Some of these functions can be re-implemented.

Additional information on these library functions is available in the `rt_heap.h`, `rt_locale.h`, `rt_misc.h`, and `rt_sys.h` include files and the `rt_memory.s` assembler file.

4.4.1 How C and C++ programs use the library functions

This section describes specific library functions that are used to initialize the execution environment and application, library exit functions, and target-dependent library functions that the application itself might call during its execution.

Initializing the execution environment and executing the application

The entry point of a program is at `__main` in the C library where library code does the following:

1. Copies non-root execution regions from their load addresses to their execution addresses.
2. Zeroes ZI regions.
3. Branches to `__rt_entry`.

If you do not want the library to do this, you can define your own `__main` that simply branches to `__rt_entry` as in Example 4-1.

Example 4-1

```
    IMPORT __rt_entry
    EXPORT __main
    ENTRY
__main
    B      __rt_entry
    END
```

The library function `__rt_entry()` runs the program as follows:

1. Calls `__rt_stackheap_init()` to set up the stack and heap.
2. Calls `__rt_lib_init()` to initialize referenced library functions, initialize the locale and, if necessary, set up `argc` and `argv` for `main()`.
3. Calls `main()`, the user-level root of the application.

   From `main()`, your program might call, among other things, library functions. See *Library functions called from main()* on page 4-21 for more information.
4. Calls `exit()` with the value returned by `main()`.

Library functions called from main()

The function `main()` is the user-level root of the application and expects the execution environment to be initialized and that input/output functions can be called. While in `main()` the program might perform one of the following actions that calls user-customizable functions in the C library:

- Extend the stack or heap. See *Tailoring the run-time memory model* on page 4-60.
- Call library functions that require a callout to a user-defined function, `__rt_fp_status_addr` or `clock` for example. See *Tailoring other C library functions* on page 4-76.
- Call library functions that use `LOCALE` or `CTYPE`. See *Tailoring locale and CTYPE* on page 4-24.
- Perform floating-point calculations that require the fpu or fp library.
- Input or output directly through low-level functions, `putc` for example, or indirectly through high-level input/output functions and input/output support functions, `fprintf` or `sys_open` for example. See *Tailoring the input/output functions* on page 4-67.
- Raise an error or other signal, `ferror` for example. See *Tailoring error signalling, error handling, and program exit* on page 4-47.

4.4.2 Exiting from the program

The program can exit normally at the end of `main()` or it can exit prematurely due to an error.

Exiting from an assert

The exit sequence from an assert is:

1. `assert()` calls `abort()`.
2. `abort()` calls `__rt_raise()`.
3. If `__rt_raise()` returns, `abort()` tries to finalize the library.

If you are creating an application that does not use the library, `assert()` will work if you retarget `abort()`. However, `abort()` does not have to finalize the library. The ANSI standard states that finalization is an implementation decision.

One solution for retargeting is to replace the `__rt_exit` call in `abort()` with a call to `_sys_exit()`. The function `assert()` would then work after retargeting only `_sys_exit`.

Application exit from __rt_entry()

If you replace `__rt_entry()` with your own function, it must end with a call to one of the following functions:

`exit()`	To get full `atexit()` handling and library shut down
`__rt_exit()`	To correctly shut down the library, bypassing `atexit()` processing
`_sys_exit()`	To exit directly to the execution environment, bypassing `atexit()`.

4.5 Tailoring static data access

This section describes using callouts from the C library to access static data. There are three types of C library function with regard to the use of static data:

- functions that do not use any static data of any kind, for example `fprintf()`
- functions that manage a static state, for example `malloc()`, `rand()`, and `strtok()`
- functions that do not manage a static state, but use static data in a way that is specific to their ARM implementation, for example `isalpha()`.

When the C library does something that requires implicit static data, it uses a callout to a function you can replace. These functions are shown in Table 4-4:

Table 4-4 Callouts

Function	Description
`__rt_errno_addr()`	Called to get the address of the variable `errno`. See *__rt_errno_addr()* on page 4-49.
`__rt_fp_status_addr()`	Called by the floating-point support code to get the address of the floating-point status word. See *__rt_fp_status_addr()* on page 4-51.
The `locale` functions	The function `__user_libspace()` creates a block of private static data for the library. See *Tailoring locale and CTYPE* on page 4-24.

The functions above do not use semihosting.

See also *Tailoring the run-time memory model* on page 4-60 for more information about memory use.

The default implementation of `__user_libspace()` creates a 64-byte block in the ZI segment. Even if your application does not have a `main()` function, the `__user_libspace()` function does not normally need to be redefined.

4.6 Tailoring locale and CTYPE

This section describes functions related to locale. Applications use locale when they display or process data that is dependent on the local language or region, for example character order, monetary symbols, decimal point, time, and date.

See the `rt_locale.h` include file for more information on locale-related functions.

4.6.1 Selecting locale at link time

The **`locale`** subsystem of the C library can be selected at link time or extended to be selectable at runtime. The following points describe the use of locale categories by the library:

- The default implementation of each locale category is for the C locale. The library also provides an alternative, ISO8859-1 (Latin 1 alphabet) implementation of each locale category that you can select at link time.
- Both the C and ISO8859-1 default implementations provide only one locale to select at runtime.
- You can replace each locale category individually.
- You can include as many locales in each category as you choose and you can name your locales as you wish.
- Each locale category uses one word in the private static data of the library.
- The locale category data is read-only and position independent.
- `scanf()` forces the inclusion of the LC_CTYPE locale category, but in either of the default locales this adds only 260 bytes of read-only data to several kilobytes of code.

Implementation

To select an ISO8859-1 (Latin-1 alphabet) locale category, include a call from your application to the functions shown in Table 4-5.

Table 4-5 Default locales

Function	Description
`__use_iso8859_ctype()`	Selects the ISO8859-1 (Latin-1) classification of characters (this is essentially 7-bit ASCII, except that the top-bit-set character codes 160-255 represent a selection of useful European punctuation characters, letters, and accented letters).
`__use_iso8859_collate()`	Selects the strcoll/strxfrm collation table appropriate to the Latin-1 alphabet. The default C locale needs no collation table.
`__use_iso8859_monetary()`	Selects the Sterling monetary category using Latin-1 coding.
`__use_iso8859_numeric()`	Selects separating thousands with commas in the printing of numeric values.
`__use_iso8859_locale()`	Selects all the above iso8859 selections.

There is no ISO8859-1 version of the LC_TIME category.

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

4.6.2 Selecting locale at run time

The C library function `setlocale()` selects a locale at runtime for the locale category, or categories, specified in its arguments. It does this by selecting the requested locale separately in each locale category. In effect, each locale category is a small filing system containing an entry for each locale.

Each locale category is processed by a function like `_get_lc_`*`category`*, for example:

```
void const *_get_lc_time (void *null, char const *locale_name)
```

`_get_lc_time()` returns the address of the `time` filing system entry for the locale named `locale_name`, or `NULL` the entry was not found.

The implementation of each locale category must supply a selection function as shown in Table 4-6.

Table 4-6 Locale categories

Function	Description
`_get_lc_ctype()`	Returns a pointer to the first element in a user-defined array that holds character attributes. See *_get_lc_ctype()* on page 4-28.
`_get_lc_collate()`	Returns a pointer to the first element in a user-defined array that holds sorting attributes. See *_get_lc_collate()* on page 4-31.
`_get_lc_monetary()`	Returns a pointer to the user-defined `__lc_monetary_blk` structure. See *_get_lc_monetary()* on page 4-34.
`_get_lc_numeric()`	Returns a pointer to the user-defined `__lc_numeric_blk` structure. See *_get_lc_numeric()* on page 4-35.
`_get_lc_time()`	Returns a pointer to the user-defined `__lc_time_blk` structure. See *_get_lc_time()* on page 4-36.

These functions are described below. C header files describing what must be implemented and providing some useful support macros, are given in `locale.h` and `rt_locale.h`.

Implementation

For each category, changing locale is achieved by changing a pointer into the read-only data for the locale category. Except for default locales, the data must be user-supplied.

All locale blocks for a category are collected into a read-only, position-independent, in-memory file system structure. The C library provides a set of macros to create the blocks and the `_findlocale()` function to search the file system.

You can define a set of run-time selectable locales by using the supplied re-implementations as a starting point. Your application will not call `_get_lc_`*`category`* functions directly. `_get_lc_`*`category`* functions are called by `setlocale()` and `rt_lib_init()`. You implement new locales by providing new locale definition blocks and re-implementations of `_get_lc_`*`category`* for `setlocale()` to use as in Example 4-2.

Example 4-2

```
void const *_get_lc_ctype(void const *null, char const *name) {
    return _findlocale(&lcctype_c_index, name);
}
```

4.6.3 Macros and utility functions

The macros and utility functions listed in Table 4-7 simplify the process of creating and using locale blocks. See the `rt_locale.h` file for more information.

Table 4-7 locale macros

Function or macro	Description
__LC_CTYPE_DEF	Use this macro to create a block of values for the character set. See *_get_lc_ctype()* on page 4-28.
__LC_COLLATE_DEF	Use this macro to create a block of sorting values for the character set. See *_get_lc_collate()* on page 4-31.
__LC_TIME_DEF	Use this macro to create a block of time formatting values. See *_get_lc_time()* on page 4-36.
__LC_NUMERIC_DEF	Use this macro to create a block of numeric formatting values. See *_get_lc_numeric()* on page 4-35.
__LC_MONETARY_DEF	Use this macro to create a block of monetary formatting values. See *_get_lc_monetary()* on page 4-34.
__LC_INDEX_END	Use this macro to declare the end of an index of formatting values. See *Using the macros* on page 4-27.
`_findlocale()`	Use this function to return the address of a locale block. See *_findlocale()* on page 4-40.

Using the macros

The data blocks for a single locale category must be contiguous and the LC_INDEX_END macro must be the last macro in the sequence.

The examples in each locale category use two test macros that are defined as:

```
#define EQI(i,j) assert(i==j)
#define EQS(s,t) assert(!strcmp(s,t))
```

4.6.4 _get_lc_ctype()

The ctype implementation is selected at link time to be either:

- The C locale only. This is the default.
- The ISO 8859 (Latin-1) locale.

You can define your own ctype attribute table with the following characteristics:

- It must be read-only.
- It is a byte array with indexes ranging from –1 to 255 inclusive (257 bytes in total)
- Each byte is interpreted as 8 attribute bits, the values are defined in `ctype.h` as follows:

__C	white-space characters
__P	punctuation characters
__B	blank characters
__L	lower-case letters
__U	upper-case letters
__N	decimal digits
__C	control characters
__X	hexadecimal-digit letters A-F and a-f.

The first element in the array, the element located at –1, must be zero. A skeletal implementation of the functions that return CTYPE data is shown in Example 4-3:

Example 4-3

```
__LC_CTYPE_DEF(lcctype_c, "C")
{
    __C, __C, __C, __C, __C, __C, __C, __C, __C,            /* 0x00-0x08 */
    __C+__S,__C+__S,__C+__S,__C+__S,__C+__S,    /* 0x09-0x0D (BS,LF,VT,FF,CR) */
    __C, __C, __C, __C, __C, __C, __C, __C, __C,            /* 0x0E-0x16 */
    __C, __C, __C, __C, __C, __C, __C, __C, __C,            /* 0x17-0x1F */
    __B+__S,                                                /* space */
    __P, __P, __P, __P, __P, __P, __P, __P,                 /* !"#$%&'( */
    __P, __P, __P, __P, __P, __P, __P,                      /* )*+,-./ */
    __N, __N, __N, __N, __N, __N, __N, __N, __N, __N,       /* 0-9 */
    __P, __P, __P, __P, __P, __P, __P,                      /* :;<=>?@ */
```

```
        __U+__X, __U+__X, __U+__X, __U+__X, __U+__X, __U+__X,    /* A-F */
        __U, __U, __U, __U, __U, __U, __U, __U, __U, __U,        /* G-P */
        __U, __U, __U, __U, __U, __U, __U, __U, __U, __U,        /* Q-Z */
        __P, __P, __P, __P, __P, __P,                            /* [\]^_` */
        __L+__X, __L+__X, __L+__X, __L+__X, __L+__X, __L+__X,    /* a-f */
        __L, __L, __L, __L, __L, __L, __L, __L, __L, __L,        /* g-p */
        __L, __L, __L, __L, __L, __L, __L, __L, __L, __L,        /* q-z */
        __P, __P, __P, __P,                                      /* {|}~ */
        __C,                                                     /* 0x7F */
        /* the whole of the top half is illegal characters */
        0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
        0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
    };
    __LC_CTYPE_DEF(lcctype_iso8859_1, "ISO8859-1")
    {
        __C, __C, __C, __C, __C, __C, __C, __C, __C,             /* 0x00-0x08 */
        __C+__S,__C+__S,__C+__S,__C+__S,__C+__S,  /* 0x09-0x0D (BS,LF,VT,FF,CR) */
        __C, __C, __C, __C, __C, __C, __C, __C, __C,             /* 0x0E-0x16 */
        __C, __C, __C, __C, __C, __C, __C, __C, __C,             /* 0x17-0x1F */
        __B+__S,                                                 /* space */
        __P, __P, __P, __P, __P, __P, __P, __P,                  /* !"#$%&'( */
        __P, __P, __P, __P, __P, __P, __P,                       /* )*+,-./ */
        __N, __N, __N, __N, __N, __N, __N, __N, __N, __N,        /* 0-9 */
        __P, __P, __P, __P, __P, __P, __P,                       /* :;<=>?@ */
        __U+__X, __U+__X, __U+__X, __U+__X, __U+__X, __U+__X,    /* A-F */
        __U, __U, __U, __U, __U, __U, __U, __U, __U, __U,        /* G-P */
        __U, __U, __U, __U, __U, __U, __U, __U, __U, __U,        /* Q-Z */
        __P, __P, __P, __P, __P, __P,                            /* [\]^_` */
        __L+__X, __L+__X, __L+__X, __L+__X, __L+__X, __L+__X,    /* a-f */
        __L, __L, __L, __L, __L, __L, __L, __L, __L, __L,        /* g-p */
        __L, __L, __L, __L, __L, __L, __L, __L, __L, __L,        /* q-z */
        __P, __P, __P, __P,                                      /* {|}~ */
        __C,                                                     /* 0x7F */
        /* ISO8859-1 top half:
         * - 0x80-0x9f are control chars
         * - 0xa0 is nonbreaking space (whitespace)
         * - 0xa1-0xbf are punctuation chars
         * - 0xc0-0xdf are uppercase chars except times sign at 0xd7
         * - 0xe0-0xff are lowercase chars except divide sign at 0xf7 */
        __C,__C,__C,__C,__C,__C,__C,__C,        /* 0x80 - 0x87 */
        __C,__C,__C,__C,__C,__C,__C,__C,        /* 0x88 - 0x8f */
        __C,__C,__C,__C,__C,__C,__C,__C,       /* 0x90 - 0x97 */
        __C,__C,__C,__C,__C,__C,__C,__C,       /* 0x98 - 0x9f */
        __B+__S,__P,__P,__P,__P,__P,__P,__P,   /* 0xa0 - 0xa7 */
        __P,__P,__P,__P,__P,__P,__P,__P,       /* 0xa8 - 0xaf */
        __P,__P,__P,__P,__P,__P,__P,__P,       /* 0xb0 - 0xb7 */
        __P,__P,__P,__P,__P,__P,__P,__P,       /* 0xb8 - 0xbf */
        __U,__U,__U,__U,__U,__U,__U,__U,       /* 0xc0 - 0xc7 */
        __U,__U,__U,__U,__U,__U,__U,__U,       /* 0xc8 - 0xcf */
        __U,__U,__U,__U,__U,__U,__U,__P,       /* 0xd0 - 0xd7 */
        __U,__U,__U,__U,__U,__U,__U,__U,       /* 0xd8 - 0xdf */
        __L,__L,__L,__L,__L,__L,__L,__L,       /* 0xe0 - 0xe7 */
        __L,__L,__L,__L,__L,__L,__L,__L,       /* 0xe8 - 0xef */
        __L,__L,__L,__L,__L,__L,__L,__P,       /* 0xf0 - 0xf7 */
        __L,__L,__L,__L,__L,__L,__L,__L,           /* 0xf8 - 0xff */
    };
    _LC_INDEX_END(lcctype_dummy)

    void const *_get_lc_ctype(void const *null, char const *name) {
        return _findlocale(&lcctype_c_index, name);
    }

    void test_lc_ctype(void) {
        EQS(setlocale(LC_CTYPE, NULL), "C");  /* verify starting point */
        EQI(!!isalpha('@'), 0);               /* test off-by-one */
        EQI(!!isalpha('A'), 1);
        EQI(!!isalpha('\xc1'), 0);            /* C locale: isalpha(Aacute)==0 */
        EQI(!setlocale(LC_CTYPE, "ISO8859-1"), 0);   /* setlocale should work */
        EQS(setlocale(LC_CTYPE, NULL), "ISO8859-1");
        EQI(!!isalpha('@'), 0);               /* test off-by-one */
        EQI(!!isalpha('A'), 1);
        EQI(!!isalpha('\xc1'), 1);            /* ISO8859 locale: isalpha(Aacute)!=0 */
        EQI(!setlocale(LC_CTYPE, "C"), 0);    /* setlocale should work */
        EQS(setlocale(LC_CTYPE, NULL), "C");
        EQI(!!isalpha('@'), 0);               /* test off-by-one */
        EQI(!!isalpha('A'), 1);
        EQI(!!isalpha('\xc1'), 0);            /* C locale: isalpha(Aacute)==0 */
    }
```

4.6.5 _get_lc_collate()

`_get_lc_collate` must return a pointer to the 0th entry in an array of unsigned bytes whose indexes range from 0 to 255 inclusive (256 bytes total).

Each element gives the position in the collation sequence of the character represented by the index of the element. For example, if you wanted `strcoll()` to sort strings beginning with `Z` in between those beginning with `A` and those beginning with `B`, you would set up the LC_COLLATE table so that `array['A'] < array['Z']` and `array['Z'] < array['B']`.

`_get_lc_collate` must return a pointer to a collate structure. Use the macros in Example 4-4 to create the structure.

Example 4-4

```
__LC_COLLATE_TRIVIAL_DEF(lccoll_c, "C")
__LC_COLLATE_DEF(lccoll_iso8859_1, "ISO8859-1")
{
    /* Things preceding letters have normal ASCII ordering */
    0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
    0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
    0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
    0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f,
    0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27,
    0x28, 0x29, 0x2a, 0x2b, 0x2c, 0x2d, 0x2e, 0x2f,
    0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37,
    0x38, 0x39, 0x3a, 0x3b, 0x3c, 0x3d, 0x3e, 0x3f,
    0x40,  /* @ */    0x41,  /* A - then 7 A variants */
    0x49,  /* B */    0x4a,  /* C - then 1 C variant */
    0x4c,  /* D */    0x4d,  /* E - then 4 E variants */
    0x52,  /* F */    0x53,  /* G */
    0x54,  /* H */    0x55,  /* I - then 4 I variants */
    0x5a,  /* J */    0x5b,  /* K */
    0x5c,  /* L */    0x5d,  /* M */
    0x5e,  /* N - then 1 N variant */
    0x60,  /* O - then 6 O variants */
    0x67,  /* P */    0x68,  /* Q */
    0x69,  /* R */    0x6a,  /* S */
    0x6b,  /* T */    0x6c,  /* U - then 4 U variants */
    0x71,  /* V */    0x72,  /* W */
    0x73,  /* X */    0x74,  /* Y - then 1 Y variant */
    0x76,  /* Z - then capital Eth & Thorn */
    0x79,  /* [ */    0x7a,  /* \ */
    0x7b,  /* ] */    0x7c,  /* ^ */
    0x7d,  /* _ */    0x7e,  /* ` */
    0x7f,  /* a - then 7 a variants */
    0x87,  /* b */    0x88,  /* c - then 1 c variant */
    0x8a,  /* d */    0x8b,  /* e - then 4 e variants */
    0x90,  /* f */    0x91,  /* g */
    0x92,  /* h */    0x93,  /* i - then 4 i variants */
    0x98,  /* j */    0x99,  /* k */
    0x9a,  /* l */    0x9b,  /* m */
    0x9c,  /* n - then 1 n variant */
    0x9e,  /* o - then 6 o variants */
    0xa5,  /* p */    0xa6,  /* q */
    0xa7,  /* r */    0xa8,  /* s - then 1 s variant */
    0xaa,  /* t */    0xab,  /* u - then 4 u variants */
    0xb0,  /* v */    0xb1,  /* w */
    0xb2,  /* x */    0xb3,  /* y - then 2 y variants */
    0xb6,  /* z - then eth & thorn */
    0xb9,  /* { */    0xba,  /* | */
    0xbb,  /* } */    0xbc,  /* ~ */
    0xbd,  /* del */
    /* top bit set control characters */
    0xbe, 0xbf, 0xc0, 0xc1, 0xc2, 0xc3, 0xc4, 0xc5,
    0xc6, 0xc7, 0xc8, 0xc9, 0xca, 0xcb, 0xcc, 0xcd,
    0xce, 0xcf, 0xd0, 0xd1, 0xd2, 0xd3, 0xd4, 0xd5,
    0xd6, 0xd7, 0xd8, 0xd9, 0xda, 0xdb, 0xdc, 0xdd,
    /* other non_alpha */
    0xde, 0xdf, 0xe0, 0xe1, 0xe2, 0xe3, 0xe4, 0xe5,
    0xe6, 0xe7, 0xe8, 0xe9, 0xea, 0xeb, 0xec, 0xed,
    0xee, 0xef, 0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5,
    0xf6, 0xf7, 0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd,
    0x42,  /* A grave */    0x43,  /* A acute */
    0x44,  /* A circumflex */
    0x45,  /* A tilde */    0x46,  /* A umlaut */
    0x47,  /* A ring */     0x48,  /* AE */
    0x4b,  /* C cedilla */  0x4e,  /* E grave */
    0x4f,  /* E acute */    0x50,  /* E circumflex */
    0x51,  /* E umlaut */   0x56,  /* I grave */
    0x57,  /* I acute */    0x58,  /* I circumflex */
    0x59,  /* I umlaut */   0x77,  /* Eth */
    0x5f,  /* N tilde */    0x61,  /* O grave */
    0x62,  /* O acute */    0x63,  /* O circumflex */
    0x64,  /* O tilde */    0x65,  /* O umlaut */
    0xfe,  /* multiply */   0x66,  /* O with line */
    0x6d,  /* U grave */    0x6e,  /* U acute */
    0x6f,  /* U circumflex */  0x70,  /* U umlaut */
    0x75,  /* Y acute */    0x78,  /* Thorn */
    0xa9,  /* german sz */  0x80,  /* a grave */
    0x81,  /* a acute */    0x82,  /* a circumflex */
    0x83,  /* a tilde */    0x84,  /* a umlaut */
    0x85,  /* a ring */     0x86,  /* ae */
    0x89,  /* c cedilla */  0x8c,  /* e grave */
    0x8d,  /* e acute */    0x8e,  /* e circumflex */
    0x8f,  /* e umlaut */   0x94,  /* i grave */
    0x95,  /* i acute */    0x96,  /* i circumflex */
    0x97,  /* i umlaut */   0xb7,  /* eth */
    0x9d,  /* n tilde */    0x9f,  /* o grave */
    0xa0,  /* o acute */    0xa1,  /* o circumflex */
    0xa2,  /* o tilde */    0xa3,  /* o umlaut */
    0xff,  /* divide  */    0xa4,  /* o with line */
    0xac,  /* u grave */    0xad,  /* u acute */
    0xae,  /* u circumflex */ 0xaf,  /* u umlaut */
    0xb4,  /* y acute */    0xb8,  /* thorn */
    0xb5   /* y umlaut */
};
__LC_INDEX_END(lccollate_dummy)

void const *_get_lc_collate(void const *null, char const *name) {
    return _findlocale(&lccoll_c_index, name);
}

void test_lc_collate(void) {
    char buf[5];

    /* test both strxfrm and strcoll here*/
    EQS(setlocale(LC_COLLATE, NULL), "C");          /* verify starting point */
    EQS((strxfrm(buf, "\xEF", 4), buf), "\xEF");
    EQI(strcoll("\xEF", "j") < 0, 0);
    EQI(!setlocale(LC_COLLATE, "ISO8859-1"), 0);    /* setlocale should work */
    EQS(setlocale(LC_COLLATE, NULL), "ISO8859-1");
    EQS((strxfrm(buf, "\xEF", 4), buf), "\x97");
    EQI(strcoll("\xEF", "j") < 0, 1);
    EQI(!setlocale(LC_COLLATE, "C"), 0);            /* setlocale should work */
    EQS(setlocale(LC_COLLATE, NULL), "C");
    EQS((strxfrm(buf, "\xEF", 4), buf), "\xEF");
    EQI(strcoll("\xEF", "j") < 0, 0);
}
```

The `__LC_COLLATE_TRIVIAL_DEF` macro defines an array that has the element value equal to its index number. `__LC_COLLATE_TRIVIAL_DEF(lccoll_c, "C")` is equivalent to the code in Example 4-5.

Example 4-5

```
__LC_COLLATE_DEF(lccoll_c, "C")
{
0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f,
...
0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5, 0xf6, 0xf7,
0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd, 0xfe, 0xff
};
```

4.6.6 _get_lc_monetary()

`_get_lc_monetary()` must return a pointer to an `__lc_monetary_blk` structure. Use the macros in Example 4-6 to create the structure.

Example 4-6

```
__LC_MONETARY_DEF(lcmonetary_c, "C",
                  "","","","","","","",
                  255,255,255,255,255,255,255,255)
__LC_MONETARY_DEF(lcmonetary_iso8859_1, "ISO8859-1",
                  "STG ", "\243", ".", ",", "\3", "", "-",
                  2, 2, 1, 0, 1, 0, 1, 2)
__LC_INDEX_END(lcmonetary_dummy)

void const *_get_lc_monetary(void const * nullpara, char const *name) {
    return _findlocale(&lcmonetary_c_index, name);
}

void test_lc_monetary(void) {
    struct lconv lc;
    /*Test changing currency string as we change locales.*/
    EQS(setlocale(LC_MONETARY, NULL), "C");         /* verify starting point */
    _get_lconv(&lc); EQS(lc.currency_symbol, "");
    EQI(!setlocale(LC_MONETARY, "ISO8859-1"), 0);   /* setlocale should work */
    EQS(setlocale(LC_MONETARY, NULL), "ISO8859-1");
    _get_lconv(&lc); EQS(lc.currency_symbol, "\243");
    EQI(!setlocale(LC_MONETARY, "C"), 0);           /* setlocale should work */
    EQS(setlocale(LC_MONETARY, NULL), "C");   _get_lconv(&lc);
    EQS(lc.currency_symbol, "");
}
```

4.6.7 _get_lc_numeric()

`_get_lc_numeric()` must return a pointer to an `__lc_numeric_blk` structure. Use the macros in Example 4-7 to create the structure.

Example 4-7

```
__LC_NUMERIC_DEF(lcnumeric_c, "C",".","","")
__LC_NUMERIC_DEF(lcnumeric_iso8859_1, "ISO8859-1",
                ".", ",", "\3")
__LC_NUMERIC_DEF(lcnumeric_fr, "fr", ",", ".", "\3")
__LC_INDEX_END(lcnumeric_dummy)

void const *_get_lc_numeric(void const *null, char const *name) {
    return _findlocale(&lcnumeric_c_index, name);
}

void test_lc_numeric(void) {
    double pi = 4*atan(1.);
    char buf[20];

    /* Test changing decimal point as we shift in and out of French
     * numeric locale. */

    EQS(setlocale(LC_NUMERIC, NULL), "C");          /* verify starting point */
    snprintf(buf, sizeof(buf), "%g", pi); EQS(buf, "3.14159");
    EQI(!setlocale(LC_NUMERIC, "ISO8859-1"), 0);    /* setlocale should work */
    EQS(setlocale(LC_NUMERIC, NULL), "ISO8859-1");
    snprintf(buf, sizeof(buf), "%g", pi); EQS(buf, "3.14159");
    EQI(!setlocale(LC_NUMERIC, "fr"), 0);           /* setlocale should work */
    EQS(setlocale(LC_NUMERIC, NULL), "fr");
    snprintf(buf, sizeof(buf), "%g", pi); EQS(buf, "3,14159");
    EQI(!setlocale(LC_NUMERIC, "C"), 0);            /* setlocale should work */
    EQS(setlocale(LC_NUMERIC, NULL), "C");
    snprintf(buf, sizeof(buf), "%g", pi); EQS(buf, "3.14159");
}
```

The offset fields are interpreted similarly to `__lc_monetary_blk`.

4.6.8 _get_lc_time()

`_get_lc_time()` must return a pointer to a `__lc_time_blk` structure. Use the macros in Example 4-8 to create the structure.

Example 4-8 Time structure

```
__LC_TIME_DEF(lctime_c, "C",
              "Sun\0Mon\0Tue\0Wed\0Thu\0Fri\0Sat",
              "Sunday\0xxx" "Monday\0xxx" "Tuesday\0xx" "Wednesday\0"
              "Thursday\0x" "Friday\0xxx" "Saturday\0",
              "Jan\0Feb\0Mar\0Apr\0May\0Jun\0Jul\0Aug\0Sep\0Oct\0Nov\0Dec",
              "January\0xx" "February\0x" "March\0xxxx" "April\0xxxx"
              "May\0xxxxxx" "June\0xxxxx" "July\0xxxxx" "August\0xxx"
              "September\0" "October\0xx" "November\0x" "December\0",
              "AM", "PM",
              "%x %X", "%d %b %Y", "%H:%M:%S")
__LC_TIME_DEF(lctime_fr, "fr",
              "dim\0lun\0mar\0mer\0jeu\0ven\0sam",
              "dimanche\0" "lundi\0xxx" "mardi\0xxx" "mercredi\0"
              "jeudi\0xxx" "vendredi\0" "samedi\0x",
              "jan\0xfev\0xmars\0avr\0xmai\0xjuin\0"
              "juil\0aout\0sep\0xoct\0xnov\0xdec\0",
              "janvier\0xx" "fevrier\0xx" "mars\0xxxxx" "avril\0xxxx"
              "mai\0xxxxxx" "juin\0xxxxx" "juillet\0xx" "aout\0xxxxx"
              "septembre\0" "octobre\0xx" "novembre\0x" "decembre\0",
              "AM", "PM", "%A, %d %B %Y, %X", "%d.%m.%y", "%H:%M:%S")
__LC_INDEX_END(lctime_dummy)

void const *_get_lc_time(void const *null, char const *name) {
    return _findlocale(&lctime_c_index, name);
}

void test_lc_time(void) {
    struct tm tm;
    char timestr[256];

    tm.tm_sec = 13;
    tm.tm_min = 13;
    tm.tm_hour = 23;
    tm.tm_mday = 12;
    tm.tm_mon = 1;
    tm.tm_year = 98;
    tm.tm_wday = 4;
    tm.tm_yday = 42;
    tm.tm_isdst = 0;
    EQS(setlocale(LC_TIME, NULL), "C");      /* verify starting point */
    strftime(timestr, sizeof(timestr), "%c", &tm);
    EQS(timestr, "12 Feb 1998 23:13:13");
    EQI(!setlocale(LC_TIME, "fr"), 0);      /* setlocale should work */
    EQS(setlocale(LC_TIME, NULL), "fr");
    strftime(timestr, sizeof(timestr), "%c", &tm);
    EQS(timestr, "jeudi, 12 fevrier 1998, 23:13:13");
    EQI(!setlocale(LC_TIME, "C"), 0);      /* setlocale should work */
    EQS(setlocale(LC_TIME, NULL), "C");
    strftime(timestr, sizeof(timestr), "%c", &tm);
    EQS(timestr, "12 Feb 1998 23:13:13");
}
```

The offset fields are interpreted similarly to `__lc_monetary_blk`.

4.6.9 _get_lconv()

`_get_lconv()` sets the components of an `lconv` structure with values appropriate for the formatting of numeric quantities.

Syntax

`void` `_get_lconv(`**`struct`** `lconv*` *`lc`*`)`

Implementation

This extension to ANSI does not use any static data. If you are building an application that must conform strictly to the ANSI C standard, use `localeconv()` instead.

Returns

The existing `lconv` structure *`lc`* is filled with formatting data.

4.6.10 localeconv()

`localeconv()` creates and sets the components of an `lconv` structure with values appropriate for the formatting of numeric quantities according to the rules of the current locale.

Syntax

`struct` `lconv * localeconv(`**`void`**`)`

Implementation

The members of the structure with type **`char`** are strings, any of which, except `decimal_point`, can point to "" to indicate that the value is not available in the current locale or is of zero length.

The members with type **`char`** are non-negative numbers. Any of the members can be `CHAR_MAX` to indicate that the value is not available in the current locale.

The members included in `lconv` are described in *The lconv structure* on page 4-45.

Returns

The function returns a pointer to the filled-in object. The structure pointed to by the return value will not be modified by the program, but might be overwritten by a subsequent call to the `localeconv()` function. In addition, calls to the `setlocale()` function with categories `LC_ALL`, `LC_MONETARY`, or `LC_NUMERIC` might overwrite the contents of the structure.

4.6.11 setlocale()

Selects the appropriate locale as specified by the *category* and *locale* arguments.

Syntax

```
char* setlocale(int category, const char* locale)
```

Implementation

The `setlocale()` function is be used to change or query part or all of the current locale. The effect of the category argument for each value is described below. A value of `"C"` for *locale* specifies the minimal environment for C translation. An empty string, `""`, for *locale* specifies the implementation-defined native environment. At program startup the equivalent of `setlocale(LC_ALL, "C")` is executed.

The values of *category* are:

LC_COLLATE	Affects the behavior of `strcoll()`.
LC_CTYPE	Affects the behavior of the character handling functions.
LC_MONETARY	Affects the monetary formatting information returned by `localeconv()`.
LC_NUMERIC	Affects the decimal-point character for the formatted input/output functions and the string conversion functions and the numeric formatting information returned by `localeconv()`.
LC_TIME	Could affect the behavior of `strftime()`. For currently supported locales, the option has no effect.
LC_ALL	Affects all locale categories. This is the bitwise OR of the above categories.

Returns

If a pointer to string is given for *locale* and the selection can be honoured, the string associated with the specified category for the new locale is returned. If the selection can not be honoured, a null pointer is returned and the locale is not changed.

A null pointer for *locale* causes the string associated with the category for the current locale to be returned and the locale is not changed.

If *category* is LC_ALL and the most recent successful locale-setting call used a category other than LC_ALL a composite string may need to be returned. The string returned is such that a subsequent call with that string and its associated category will restore that part of the program's locale. The string returned will not be modified by the program, but might be overwritten by a subsequent call to `setlocale()`.

4.6.12 _findlocale()

`findlocale()` searches the locale database and returns a pointer to the data block for the requested category and locale.

Syntax

void const* _findlocale(**void const*** *index*, **char const** **name*)

Returns

Returns a pointer to the requested data block.

4.6.13 __LC_CTYPE_DEF

This macro is used to create CTYPE blocks. The definition from `rt_locale.h` and sample code are shown in Example 4-9.

Example 4-9

```
#define __LC_CTYPE_DEF(sym,ln) \
static const int sym##_index = ~3 & (3 + (268+(~3 & (3 + sizeof(ln))))); \
static const char sym##_lname[~3 & (3 + sizeof(ln))] = ln; \
static const int sym##_pname = -4-(~3 & (3 + sizeof(ln))); \
static const char sym##_start = 0; \
static const char sym##_table[256] =
```

For all the macros, the first two arguments are a symbol prefix and a locale name. The resulting locale block will be addressed by the expression `&symprefix_start`, and the index entry by the expression `&symprefix_index`.

Usage

See *_get_lc_ctype()* on page 4-28.

4.6.14 __LC_COLLATE_DEF

This macro is used to create collate blocks used when sorting ASCII characters. The definition from `rt_locale.h`, the definition of a macro for creating an empty table, and sample code are shown in Example 4-10.

For all the macros, the first two arguments are a symbol prefix and a locale name. The resulting locale block will be addressed by the expression `&symprefix_start`, and the index entry by the expression `&symprefix_index`.

Example 4-10 Macro for use with array

```
#define __LC_COLLATE_DEF(sym,ln) \
static const int sym##_index = ~3&(3+(268+(~3&(3+sizeof(ln))))); \
static const char sym##_lname[~3 & (3 + sizeof(ln))] = ln; \
static const int sym##_pname = -4-(~3 & (3 + sizeof(ln))); \
static const int sym##_start = 4; \
static const char sym##_table[] =
```

Example 4-11 Macro that generates default table

```
#define __LC_COLLATE_TRIVIAL_DEF(sym,ln) \
static const int sym##_index = ~3&(3+(12+(~3&(3+sizeof(ln))))); \
static const char sym##_lname[~3 & (3 + sizeof(ln))] = ln; \
static const int sym##_pname = -4-(~3 & (3 + sizeof(ln))); \
static const int sym##_start = 0;
```

Usage

See *_get_lc_collate()* on page 4-31.

4.6.15 __LC_TIME_DEF

This macro is used to create blocks used when formatting time or date values. The definition from `rt_locale.h` and sample code are shown in Example 4-12.

For all the macros, the first two arguments are a symbol prefix and a locale name. The resulting locale block will be addressed by the expression `&symprefix_start`, and the index entry by the expression `&symprefix_index`.

Example 4-12

```
#define __LC_TIME_DEF(sym,ln,wa,wf,ma,mf,am,pm,dt,df,tf) \
static const int sym##_index = ~3 & (3 + (sizeof(wa)+sizeof(wf)+sizeof(ma)+ \
sizeof(mf)+sizeof(am)+sizeof(pm)+ \
sizeof(dt)+sizeof(df)+sizeof(tf)+ \
60+(~3 & (3 + sizeof(ln))))); \
static const char sym##_lname[~3 & (3 + sizeof(ln))] = ln; \
static const int sym##_pname = -4-(~3 & (3 + sizeof(ln))); \
static const int sym##_start = 52; \
static const int sym##_wfoff = (sizeof(wa)+52); \
static const int sym##_maoff = (sizeof(wa)+sizeof(wf)+52); \
static const int sym##_mfoff = (sizeof(wa)+sizeof(wf)+sizeof(ma)+52); \
static const int sym##_amoff = (sizeof(wa)+sizeof(wf)+sizeof(ma)+ \
sizeof(mf)+52); \
static const int sym##_pmoff = (sizeof(wa)+sizeof(wf)+sizeof(ma)+ \
sizeof(mf)+sizeof(am)+52); \
static const int sym##_dtoff = (sizeof(wa)+sizeof(wf)+sizeof(ma)+ \
sizeof(mf)+sizeof(am)+sizeof(pm)+52); \
static const int sym##_dfoff = (sizeof(wa)+sizeof(wf)+sizeof(ma)+ \
sizeof(mf)+sizeof(am)+sizeof(pm)+ \
sizeof(dt)+52); \
static const int sym##_tfoff = (sizeof(wa)+sizeof(wf)+sizeof(ma)+ \
sizeof(mf)+sizeof(am)+sizeof(pm)+ \
sizeof(dt)+sizeof(df)+52); \static const int sym##_wasiz = (sizeof(wa)/7); \
static const int sym##_wfsiz = (sizeof(wf)/7); \
static const int sym##_masiz = (sizeof(ma)/12); \
static const int sym##_mfsiz = (sizeof(mf)/12); \
static const char sym##_watxt[] = wa; \
static const char sym##_wftxt[] = wf; \
static const char sym##_matxt[] = ma; \
static const char sym##_mftxt[] = mf; \
static const char sym##_amtxt[] = am; \
static const char sym##_pmtxt[] = pm; \
static const char sym##_dttxt[] = dt; \
static const char sym##_dftxt[] = df; \
static const char sym##_tftxt[] = tf;
```

Usage

See *_get_lc_time()* on page 4-36.

4.6.16 __LC_NUMERIC_DEF

This macro is used to create blocks used when formatting numbers. The definition from `rt_locale.h` and sample code are shown in Example 4-13.

For all the macros, the first two arguments are a symbol prefix and a locale name. The resulting locale block will be addressed by the expression `&symprefix_start`, and the index entry by the expression `&symprefix_index`.

Example 4-13

```
#define __LC_NUMERIC_DEF(sym,ln,dp,ts,gr) \
static const int sym##_index = ~3 & (3 + (sizeof(dp)+sizeof(ts)+sizeof(gr)+ \
20) + (~3 & (3 + sizeof(ln)))); \
static const char sym##_lname[~3 & (3 + sizeof(ln))] = ln; \
static const int sym##_pname = -4-(~3 & (3 + sizeof(ln))); \
static const int sym##_start = 12; \
static const int sym##_tsoff = (sizeof(dp)+12); \
static const int sym##_groff = (sizeof(dp)+sizeof(ts)+12); \
static const char sym##_dptxt[] = dp; \
static const char sym##_tstxt[] = ts; \
static const char sym##_grtxt[] = gr;
```

Usage

See *_get_lc_numeric()* on page 4-35.

4.6.17 __LC_MONETARY_DEF

This macro is used to create blocks used when formatting monetary values. The definition from `rt_locale.h` and sample code are shown in Example 4-14.

For all the macros, the first two arguments are a symbol prefix and a locale name. The resulting locale block will be addressed by the expression `&symprefix_start`, and the index entry by the expression `&symprefix_index`.

Example 4-14

```
#define __LC_MONETARY_DEF(sym,ln,ic,cs,md,mt,mg,ps,ns, \
                          id,fd,pc,pS,nc,nS,pp,np) \
static const int sym##_index = ~3 & (3 + (sizeof(ic)+sizeof(cs)+sizeof(md)+ \
                                          sizeof(mt)+sizeof(mg)+sizeof(ps)+ \
                                          sizeof(ns)+44) \
                                     + (~3 & (3 + sizeof(ln)))); \
static const char sym##_lname[~3 & (3 + sizeof(ln))] = ln; \
static const int sym##_pname = -4-(~3 & (3 + sizeof(ln))); \
static const char sym##_start = id; \
static const char sym##_fdchr = fd; \
static const char sym##_pcchr = pc; \
static const char sym##_pSchr = pS; \
static const char sym##_ncchr = nc; \
static const char sym##_nSchr = nS; \
static const char sym##_ppchr = pp; \
static const char sym##_npchr = np; \
static const int sym##_icoff = 36; \
static const int sym##_csoff = (sizeof(ic)+36); \
static const int sym##_mdoff = (sizeof(ic)+sizeof(cs)+36); \
static const int sym##_mtoff = (sizeof(ic)+sizeof(cs)+sizeof(md)+36); \
static const int sym##_mgoff = (sizeof(ic)+sizeof(cs)+sizeof(md)+ \
                                sizeof(mt)+36); \
static const int sym##_psoff = (sizeof(ic)+sizeof(cs)+sizeof(md)+ \
                                sizeof(mt)+sizeof(mg)+36); \
static const int sym##_nsoff = (sizeof(ic)+sizeof(cs)+sizeof(md)+ \
                                sizeof(mt)+sizeof(mg)+sizeof(ps)+36); \
static const char sym##_ictxt[] = ic; \
static const char sym##_cstxt[] = cs; \
static const char sym##_mdtxt[] = md; \
static const char sym##_mttxt[] = mt; \
static const char sym##_mgtxt[] = mg; \
static const char sym##_pstxt[] = ps; \
static const char sym##_nstxt[] = ns;
```

Usage

See *_get_lc_monetary()* on page 4-34.

4.6.18 __LC_INDEX_END

This macro is used to declare the end of an index. `symprefix` is provided to ensure a unique name. The definition from `rt_locale.h` and sample code are shown in Example 4-15.

Example 4-15

```
#define __LC_INDEX_END(symprefix)  static const int symprefix##_index = 0;
```

4.6.19 The lconv structure

The lconv structure contains numeric formatting information. The structure is filled by the functions _get_lconv() and localeconv(). The setlocale() function must be called to initialize the lconv structure prior to using the structure in any other functions.

The definition of lconv from locale.h is shown in Example 4-16.

Example 4-16 lconv structure

```
struct lconv {
  char *decimal_point;
       /* The decimal point character used to format non-monetary quantities */
  char *thousands_sep;
       /* The character used to separate groups of digits to the left of the */
       /* decimal point character in formatted non-monetary quantities.      */
  char *grouping;
       /* A string whose elements indicate the size of each group of digits  */
       /* in formatted non-monetary quantities. See below for more details.  */
  char *int_curr_symbol;
       /* The international currency symbol applicable to the current locale.*/
       /* The first three characters contain the alphabetic international    */
       /* currency symbol in accordance with those specified in ISO 4217.    */
       /* Codes for the representation of Currency and Funds. The fourth     */
       /* character (immediately preceding the null character) is the        */
       /* character used to separate the international currency symbol from  */
       /* the monetary quantity.                                             */
  char *currency_symbol;
       /* The local currency symbol applicable to the current locale.        */
  char *mon_decimal_point;
       /* The decimal-point used to format monetary quantities.              */
  char *mon_thousands_sep;
       /* The separator for groups of digits to the left of the decimal-point*/
       /* in formatted monetary quantities.                                  */
  char *mon_grouping;
       /* A string whose elements indicate the size of each group of digits  */
       /* in formatted monetary quantities. See below for more details.      */
  char *positive_sign;
       /* The string used to indicate a non-negative-valued formatted        */
       /* monetary quantity.                                                 */
  char *negative_sign;
       /* The string used to indicate a negative-valued formatted monetary   */
       /* quantity.                                                          */
  char int_frac_digits;
       /* The number of fractional digits (those to the right of the         */
       /* decimal-point) to be displayed in an internationally formatted     */
       /* monetary quantities.                                                */
  char frac_digits;
       /* The number of fractional digits (those to the right of the         */
       /* decimal-point) to be displayed in a formatted monetary quantity.   */
  char p_cs_precedes;
       /* Set to 1 or 0 if the currency_symbol respectively precedes or      */
       /* succeeds the value for a non-negative formatted monetary quantity. */
  char p_sep_by_space;
       /* Set to 1 or 0 if the currency_symbol respectively is or is not     */
       /* separated by a space from the value for a non-negative formatted   */
       /* monetary quantity.                                                 */
  char n_cs_precedes;
       /* Set to 1 or 0 if the currency_symbol respectively precedes or      */
       /* succeeds the value for a negative formatted monetary quantity.     */
  char n_sep_by_space;
       /* Set to 1 or 0 if the currency_symbol respectively is or is not     */
       /* separated by a space from the value for a negative formatted       */
       /* monetary quantity.                                                 */
  char p_sign_posn;
       /* Set to a value indicating the position of the positive_sign for a  */
       /* non-negative formatted monetary quantity. See below for more details*/
  char n_sign_posn;
       /* Set to a value indicating the position of the negative_sign for a  */
       /* negative formatted monetary quantity. */
};
```

The elements of `grouping` and `non_grouping` are interpreted as follows:

CHAR_MAX No further grouping is to be performed.

0 The previous element is repeated for the remainder of the digits.

other The value is the number of digits that compromise the current group. The next element is examined to determine the size of the next group of digits to the left of the current group.

The value of `p_sign_posn` and `n_sign_posn` is interpreted as follows:

0 Parentheses surround the quantity and currency symbol.

1 The sign string precedes the quantity and currency symbol.

2 The sign string is after the quantity and currency symbol.

3 The sign string immediately proceeds the currency symbol.

4 The sign string immediately succeeds the currency symbol.

4.7 Tailoring error signalling, error handling, and program exit

All trap or error signals raised by the C library go through the `__raise()` function. You can re-implement this function or the lower-level functions that it uses.

Caution

The IEEE 754 standard for floating-point processing states that the default response to an exception is to proceed without a trap. You can modify floating-point error handling by tailoring the functions and definitions in `fenv.h`. See also Chapter 8 *Floating-point Support*.

See the `rt_misc.h` include file for more information on error-related functions.

The trap and error-handling functions are shown in Table 4-8.

Table 4-8 Trap and error handling

Function	Description
`_sys_exit()`	Called, eventually, by all exits from the library. See *_sys_exit()* on page 4-48.
`errno`	Is a static variable used with error handling. See *errno* on page 4-48.
`__raise()`	Raises a signal to indicate a runtime anomaly. See *__raise()* on page 4-49.
`__rt_errno_addr()`	This function is called to obtain the address of the C library. See *__rt_errno_addr()* on page 4-49.
`__rt_fp_status_addr()`	This function is called to obtain the address of the fp status word. See *__rt_fp_status_addr()* on page 4-51.
`__default_signal_handler()`	Displays an error indication to the user. See *__default_signal_handler()* on page 4-50.
`_ttywrch()`	The default implementation of `_ttywrch` is semihosted and therefore it uses the semihosting SWI. See *_ttywrch()* on page 4-51.

4.7.1 _sys_exit()

The library exit function. All exits from the library eventually call `_sys_exit()`.

Syntax

```
void _sys_exit(int return_code)
```

Implementation

This function must not return. You can intercept application exit at a higher level by either:

- Implementing the C library function `exit()` as part of your application. You will lose `atexit()` processing and library shutdown if you do this.
- Implementing the function `__rt_exit(int n)` as part of your application. You will lose library shutdown if you do this, but `atexit()` processing will still be performed when `exit()` is called or `main()` returns.

Returns

The return code is advisory. An implementation might attempt to pass it to the execution environment.

4.7.2 errno

The C library `errno` variable is defined in the implicit static data area of the library. This area is identified by `__user_libspace()`. It occupies part of initial stack space used by the functions that established the runtime stack. The definition of `errno` is:

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

You can define `__rt_errno_addr()` if you want to place `errno` at a user-defined location instead of the default location identified by `__user_libspace()`.

Returns

The default implementation is a veneer on `__user_libspace()` that returns the address of the status word. A suitable default definition is given in the C library standard headers.

4.7.3 __rt_errno_addr()

This function is called to obtain the address of the C library `errno` variable when the C library attempts to read or write `errno`. A default implementation is provided by the library. It is very unlikely that you would have to re-implement this function.

Syntax

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

4.7.4 __raise()

This function raises a signal to indicate a runtime anomaly.

Syntax

```
int __raise(int major, int minor)
```

major	Is an integer that holds the signal number.
minor	Is an integer or string constant or variable.

Implementation

This function calls the normal C signal mechanism or the default signal handler. See also *_ttywrch()* on page 4-51 for more information.

You can replace the `__raise()` function by defining:

```
int __raise(int signal, int argument)
```

This allows you to bypass the C signal mechanism and its data-consuming signal handler vector, but otherwise gives essentially the same interface as:

```
void __default_signal_handler(int signal, int arg)
```

Returns

There are three possibilities for `__raise()` return condition:

no return	The handler performs a long jump or restart.
0	The signal was handled.
non-0	The calling code should pass that return value to the exit code. The default library implementation calls `_sys_exit(rc)` if `__raise()` returns a non-zero return code *rc*.

4.7.5 __rt_raise()

This function raises a signal to indicate a runtime anomaly.

Syntax

void `__rt_raise(`**int** *signal*, **int** *type*`)`

signal	Is an integer that holds the signal number.
type	Is an integer or string constant or variable.

Implementation

This function calls `__raise()`. See *__raise()* on page 4-49 for more information.

Depending on the value returned from `__raise()`:

no return	The handler performed a long jump or restart and `__rt_raise()` does not regain control.
0	The signal was handled and `__rt_raise()` exits.
non-0	The default library implementation calls `_sys_exit(rc)` if `__raise()` returns a non-zero return code *rc*.

4.7.6 __default_signal_handler()

This function handles a raised signal. The default action is to print an error message and exit.

Syntax

void `__default_signal_handler(`**int** *signal*, **int** *arg*`)`

Implementation

The default signal handler uses `_ttywrch()` to print a message and calls `_sys_exit()` to exit.You can replace the default signal handler by defining:

```
void __default_signal_handler(int signal, int argument)
```

4.7.7 _ttywrch()

This function writes a character to the console. The console might have been redirected. This function may be used as a last resort error handling routine.

Syntax

```
void _ttywrch(int ch)
```

Implementation

The default implementation of this function uses the semihosting SWI.

You can redefine this function, or `__raise()`, even if there is no other input/output. For example, it might write an error message to a log kept in non-volatile memory.

4.7.8 __rt_fp_status_addr()

This function returns the address of the floating-point status register.

Syntax

```
unsigned* _rt_fp_status_addr(void)
```

Implementation

If `__rt_fp_status_addr()` is not defined, the default implementation from the C library is used. The value is initialized when `__rt_lib_init()` calls `_fp_init()`. The constants for the status word are listed in `fenv.h`. The default fp status is 0.

4.8 Tailoring storage management

This section describes the functions from `rt_heap.h` that you can define if you are tailoring memory management. There are also two helper functions that you can call from your heap implementation.

See the `rt_heap.h` and `rt_memory.h` include files for more information on memory-related functions.

4.8.1 Support for malloc

`malloc()`, `realloc()`, `calloc()`, and `free()` are built on a heap abstract data type. You can either:

- Choose between Heap1 or Heap2, the two provided heap implementations.
- Write your own heap implementation of the abstract data type for heap. See *Creating your own storage-management system* on page 4-54.

Heap1: Standard heap implementation

Heap1, the default implementation, implements the smallest and simplest heap manager. The heap is managed as a singly-linked list of free blocks held in increasing address order. The allocation policy is first-fit by address.

This implementation has low overheads, but the cost of `malloc()` or `free()` grows linearly with the number of free blocks. The smallest block that can be allocated is 8 bytes. If you expect more than 100 unallocated blocks you should use Heap2.

Heap2: Alternative heap implementation

Heap2 provides a compact implementation with the cost of `malloc()` or `free()` growing logarithmically with the number of free blocks. The allocation policy is first-fit by address. The smallest block that can be allocated is 8 bytes and there is an overhead of 4 bytes.

Heap2 is recommended when near constant-time performance is required in the presence of hundreds of free blocks. To select the alternative standard implementation, `IMPORT` from assembly language, or call from C, the symbol

```
 __use_realtime_heap()
```

You can also define your own heap implementation. See *Creating your own storage-management system* on page 4-54 for more information.

Using Heap2

The Heap2 real-time heap implementation needs to know how much address space the heap will span. The smaller the address range, the more efficient the algorithm will be.

By default, the heap extent is taken to be 16MB starting at the beginning of the heap (defined as the start of the first chunk of memory given to the heap manager by `__rt_initial_stackheap()` or `__rt_heap_extend()`).

The heap bounds are given by:

```
struct __heap_extent {
       unsigned base, range;};
__value_in_regs struct __heap_extent __user_heap_extent(
       unsigned defaultbase, unsigned defaultsize);
```

The function prototype for `__user_heap_extent()` is in `rt_misc.h`.

The Heap1 algorithm does not require the bounds on the heap extent, therefore it never calls this function.

You will need to redefine `__user_heap_extent()` if:

- You require a heap to span more than 16MB of address space.
- Your memory model is able to supply a block of memory at a lower address than the first one supplied.
- You know in advance that the address space bounds of your heap are small. (In this case it is not necessary to redefine `__user_heap_extent()`, but it does speed up the heap algorithms if you do.)

The input parameters are the default values that would be used if this routine were not defined. You can, for example, leave the default base value unchanged and just adjust the size.

Note

The size field returned must be a power of two. You can set your heap extent to 4GB by returning zero for `size`.

Using a heap implementation from bare machine C

To use a heap implementation in an application that does not define `main()` and does not initialize the C library:

1. Call `_init_alloc(base, top)` to define the base and top of the memory you want to manage as a heap.
2. Define the function `unsigned __rt_heap_extend(unsigned size, void ** block)` to handle calls to extend the heap when it becomes full.

alloca()

`alloca()` behaves identically to `malloc()` except that `alloca()` has automatic garbage collection (see *alloca()* on page 4-93).

4.8.2 Creating your own storage-management system

You can implement the heap functions in Table 4-9 to create a new storage-management system.

Table 4-9 Heap functions

Function	Description
`__Heap_Descriptor`	You must define your own implementation of the abstract data type for heap. See *__Heap_Descriptor* on page 4-55.
`__Heap_Initialize()`	Initializes the heap. See *__Heap_Initialize()* on page 4-55
`__Heap_DescSize()`	Returns the size of the `__Heap_Descriptor` structure. See *__Heap_DescSize()* on page 4-56
`__Heap_ProvideMemory()`	Called to increase the size of the heap. See *__Heap_ProvideMemory()* on page 4-56
`__Heap_Alloc()`	Allocates memory from the heap to the application. See *__Heap_Alloc()* on page 4-57
`__Heap_Free()`	Returns previously allocated space to the heap. See *__Heap_Free()* on page 4-57
`__Heap_Realloc()`	Adjusts the size of an already allocated block. See *__Heap_Realloc()* on page 4-57

Table 4-9 Heap functions (Continued)

Function	Description
`__Heap_Stats()`	Called from `__heapstats()` to print statistics about the state of the heap. See *__Heap_Stats()* on page 4-58
`__Heap_Valid()`	Called to perform a consistency check on the heap. See *__Heap_Valid()* on page 4-58
`__Heap_Full()`	Attempts to acquire a new block from the system. You must not re-implement this function. See *__Heap_Full()* on page 4-59
`__Heap_Broken()`	Called when an inconsistency in the heap is detected. See *__Heap_Broken()* on page 4-59

4.8.3 __Heap_Descriptor

You must define your own implementation of the abstract data type for heap. A C header file describing this abstract data type is provided in `rt_heap.h`. You must provide the interior definition of the structure so that the other functions can find the heap data. Typical contents are given in Example 4-17.

Example 4-17

```
struct __Heap_Descriptor {
  void *my_first_free_block;
  void *my_heap_limit;
}
```

Your heap descriptor is set by `__Heap_Initialize()` and is passed to the other heap functions, for example `__Heap_Alloc()` and `__Heap_Free()`.

4.8.4 __Heap_Initialize()

Initializes the heap.

Syntax

```
void __Heap_Initialize( struct __Heap_Descriptor*h)
```

Implementation

This is called at initialization. You should redefine it to set up the fields in your heap descriptor structure to correct initial values. A typical linked-list heap would just initialize the *first_free_block* pointer to `NULL` to indicate that there are no free blocks in the heap.

4.8.5 __Heap_DescSize()

Returns the size of the `__Heap_Descriptor` structure.

Syntax

```
int __Heap_DescSize(int 0)
```

Implementation

This is called at initialization. It should return the size of your heap descriptor structure. In almost all cases the implementation in Example 4-18 is sufficient.

Example 4-18

```
extern int __Heap_DescSize(int zero) {return sizeof(__Heap_Descriptor);}
```

This routine is needed so that the library initialization can find an initial piece of memory big enough to be the heap descriptor.

4.8.6 __Heap_ProvideMemory()

Called to increase the size of the heap.

Syntax

```
void __Heap_ProvideMemory(struct __Heap_Descriptor* h, void*
base, size_t size)
```

Implementation

This is called when the system provides a chunk of memory for use by the heap. The parameters are your heap descriptor, a pointer to the new block of memory, and the size of the block. A typical `__Heap_ProvideMemory()` implementation might set up the new block of memory as a free-list entry and add it to the free chain.

4.8.7 __Heap_Alloc()

Allocates memory from the heap to the application.

Syntax

void `__Heap_Alloc( struct __Heap_Descriptor*` *h*, `size_t` *size*)

Implementation

This is called from `malloc()`, and should return a pointer to *size* bytes of memory allocated from the heap, or `NULL` if nothing can be allocated. You must ensure that the size of the block can be determined when it is time to free it. The returned block size is typically stored in the word just before its start address.

4.8.8 __Heap_Free()

Returns previously allocated space to the heap.

Syntax

void `__Heap_Free( struct __Heap_Descriptor*` *h*, **void*** *_blk*)

Implementation

This is called from `free()`, and given a pointer that was previously returned from either `__Heap_Alloc()` or `__Heap_Realloc()`. It should return the previously allocated space to the collection of free blocks in the heap.

4.8.9 __Heap_Realloc()

Adjusts the size of an already allocated block.

Syntax

void `__Heap_Realloc( struct __Heap_Descriptor*` *h*, **void*** *_blk*, `size_t` *size*)

Implementation

This is called from `realloc()`. It is never passed trivial cases such as *blk* equal to `NULL` or *size* equal to zero. It should adjust the size of the allocated block *blk* to become *size*. The reallocation might involve moving the block, copying as much of the data as is common to the old and new sizes, and returning the new address.

4.8.10 __Heap_Stats()

Called from `__heapstats()` to print statistics about the state of the heap.

Syntax

```
void *__Heap_Stats(__Heap_Descriptor *h, int(*print) (void*,
char const *format,...), void *printparam)
```

Implementation

It should output its results using the supplied printf-type *print* routine, by calls of the form:

```
print(printparam, "%d free blocks\n", nblocks);
```

The format of the statistics data is implementation-defined, so it can do nothing. This routine is effectively optional, since it will never be called unless the user program calls `__heapstats()`.

4.8.11 __Heap_Valid()

Called from `__heapvalid()` to perform a consistency check on the heap data structures and attempt to spot an invalid or corrupted heap.

Syntax

```
int __Heap_Valid(struct __Heap_Descriptor *h, int(*print)
(void*, char const *format,...), void *printparam, int verbose)
```

Implementation

It should output error messages and diagnostics using the supplied printf-type *print* routine. For example, by a call of the form:

```
print(printparam, "free block at %p is corrupt\n",block_addr);
```

This routine is effectively optional, since it will never be called unless the user program calls `__heapvalid()`.

Returns

The function should return non-zero if the heap is valid or zero if the heap is corrupted. It should use *print* to output error messages if it finds problems in the heap. If the *verbose* parameter is non-zero, it can also output diagnostic data.

4.8.12 __Heap_Full()

Attempts to acquire a new block of at least *size* bytes from the system. You must not reimplement this function.

Syntax

```
int __Heap_Full(struct __Heap_Descriptor *h, size_t size)
```

Implementation

If `__Heap_Alloc()` or `__Heap_Realloc()` cannot allocate a block of the required size from the memory owned by the heap, then before giving up and returning `NULL`, they can try calling this routine.

Remember to allow space for heap housekeeping data. If the user asks for 1000 bytes and you store a word before every allocated block, you need to ask `__Heap_Full()` for 1004 bytes, not 1000.

Before calling `__Heap_Full()`, you must ensure that the heap data structures are in a consistent state so that `__Heap_ProvideMemory()` calls will be able to add the new block to the heap successfully.

Returns

If `__Heap_Full()` is successful, it will call `__Heap_ProvideMemory()` to add the new block to the heap, and return non-zero. If it fails, it will return 0.

4.8.13 __Heap_Broken()

Called when an inconsistency in the heap is detected. You must not reimplement this function.

Syntax

```
int __Heap_Full(struct __Heap_Descriptor *h)
```

Implementation

If `__Heap_Alloc()`, `__Heap_Realloc()`, `__Heap_Free()` or `__Heap_ProvideMemory()` detect an inconsistency in the heap structures they can call this function to terminate the program with a suitable error message.

4.9 Tailoring the run-time memory model

This section describes:

- the management of writable memory by the C library as static data, heap, and stack
- functions that can be redefined in order to change how writable memory is managed.

4.9.1 The memory models

There are two managed memory models that you can select:

Single memory region

The stack grows downward from the top of the memory region while the heap grows upwards from the bottom of the region. This is the default.

Two memory regions

One memory region is for the stack and the other is for the heap. The size of heap region can be zero. The stack region can be in allocated memory or inherited from the execution environment.

To use the two-region model rather than the default one-region model, `IMPORT` (from ARM assembly language) or call (from C or C++) the symbol `__use_two_region_memory`.

The reference to the symbol `__use_two_region_memory` is used by the linker to select the memory model. There is no action taken when the code is executed. The reference to the symbol can therefore be located anywhere in the source, immediately after `main()` for example.

——— **Caution** ———

If you use the two-region memory model and do not provide any heap memory, you cannot call `malloc()`, use `stdio`, or get command-line arguments for `main()`.

If you set the size of the heap region to zero and define `__user_heap_extend()` as a function that can extend the heap, the heap will be created when it is needed.

————————————

4.9.2 Controlling the run-time memory model

The behavior of the heap and stack manager can be modified by redefining the functions listed in Table 4-10.

Table 4-10 Memory model initialization

Function	Description
`__user_initial_stackheap()`	Returns the location of the initial heap. See *__user_initial_stackheap()* on page 4-62.
`__user_heap_extend()`	Returns the size and base address of a heap extra block. See *__user_heap_extend()* on page 4-63.
`__user_stack_slop()`	Returns the amount of extra stack. See *__user_initial_stackheap()* on page 4-62.

The hidden static data for the library is provided by `__user_libspace()`. The static data area is also used as a stack during the library initialization process. This function does not normally need to be re-implemented. See *Tailoring static data access* on page 4-23.

4.9.3 Writing your own memory model

If the provided memory models do not meet your requirements, you can write your own. A memory model must define the functions described in Table 4-11. All functions are ARM-state functions (the library takes care of entry from Thumb state if this is required). An incomplete prototype implementation for the model is provided in `rt_memory.s` located in the `public` sub-directory of the library.

Use the prototype as a starting point for your own implementation.

Table 4-11 Memory model functions

Function	Description
`__rt_stackheap_init`	Sets the application stack and initial heap. See *__rt_stackheap_init()* on page 4-64.
`__rt_heap_extend`	Returns a new block of memory to add to the heap. See *__rt_heap_extend()* on page 4-65.
`__rt_stack_postlongjmp`	Atomically sets the stack pointer and stack limit pointer to their correct values after a call to longjmp. See *__rt_stack_postlongjmp()* on page 4-66.
`__rt_stack_overflow`	Handles stack overflows. See *__rt_stack_overflow()* on page 4-65.

4.9.4 __user_initial_stackheap()

Return the locations of the initial heap and stack.

Syntax

```
struct __initial_stackheap __user_initial_stackheap(unsigned R0,
unsigned SP, unsigned R2, unsigned SL)
```

Implementation

If this function is redefined, it must:

- use no more than 64 bytes of stack
- not corrupt registers other than r0 to r3 and ip
- return in r0-r3 respectively the heap base, stack base, heap limit, and stack limit.

The values of sp and sl inherited from the environment are passed as arguments in r1 and r3, respectively. An implementation of `__user_initial_stackheap()` that uses the semihosting SWI is given by the library in module `sys_stackheap.o`.

To create a version of `__user_initial_stack_heap()` that inherits sp and sl from the execution environment and does not have a heap, set r0 and r2 to r3 and return.

The definition of `__initial_stackheap` in `rt_misc.h` is:

```
struct __initial_stackheap{
unsigned heap_base, stack_base, heap_limit, stack_limit;}
```

Returns

The values returned in r0 to r3 depend on whether the one or two region model is used:

One region	(r0,r1) is the single stack region. r1 is greater than r0. r2 and r3 are ignored.
Two regions	(r0, r2) is the initial heap and (r3, r1) is the initial stack. r2is greater than or equal to r0. r3 is less than r1.

4.9.5 __user_heap_extend()

If defined, this function returns the size and base address of a heap extension block.

Syntax

```
unsigned __user_heap_extend(int 0, unsigned requested_size, void
**base)
```

Implementation

There is no default implementation of this function. If you define this function, it must have the following characteristics:

- the returned size must be at least the requested size, or 0 denoting that the request could not be honored
- the function is subject only to ATPCS constraints
- the first argument must be zero on entry. The base is returned in the register holding this argument.
- size is measured in bytes.

4.9.6 __user_heap_extent()

If defined, this function returns the base address and maximum range of the heap.

Syntax

```
__value_in_regs  struct __heap_extent
__user_heap_extent(unsigned ignore1, unsigned ignore2)
```

Implementation

There is no default implementation of this function. The values of the parameters `ignore1` and `ignore2` are not used by the function.

4.9.7 __user_stack_slop()

If defined, this function returns the size of the extra stack your system requires below sl. The extra stack is in addition to the 256 bytes required by ATPCS. The extra space might allow an interrupt handler to execute on your stack or allow a chain of unchecked functions calls.

Syntax

```
__stack_slop __user_stack_slop(void)
```

Implementation

There is no default implementation of this function.

Returns

If you define this function, it must return the following values in registers:

r0	The amount of extra stack (measured in bytes) that must always be available so an interrupt handler can execute on the stack at an arbitrary instant.
r1	The amount of extra stack (measured in bytes) that must be available after stack overflow to support recovery from overflow.

4.9.8 __rt_stackheap_init()

This function is responsible for setting up sp and sl pointing at a valid stack, and must also return in r0 and r1 the lower and upper bounds of a chunk of memory that can be used as a heap. (It can decline to do the latter, by returning r0 equal to r1. In this case, the first call to `malloc` will result in a call to `__rt_heap_extend`, described in *__rt_heap_extend()* on page 4-65.) An incomplete prototype implementation is in `rt_memory.s`. Because it is the first function called from entry, it need not preserve any other registers. On entry to this function, sp and sl are exactly as they were on entry to the whole application, so a valid stack can be inherited from the execution environment if desired. (sl is only required if stack checking is used.)

4.9.9 __rt_stack_overflow()

This function is called if a stack overflow occurs. An incomplete prototype implementation is in `rt_memory.s`

Implementation

This function is called with ip equal to the desired new sp, and with sp up to 256 bytes below sl as well.

If your memory model is used only with non-stack-checked ATPCS, you do not have to implement this function.

The stack overflow routines are called at function entry if a stack limit check fails. These are subject to the usual register-use restrictions on stack overflow routines (in particular, they cannot use r0-r3 because the arguments are still held there, and they cannot use v-registers in case the routine did not save them).

Returns

The function does not return to lr. It must return by branching to `__rt_stack_overflow_return`.

4.9.10 __rt_heap_extend()

This function should return a new block of memory to add to the heap (if that is possible). If you reimplement the other memory model functions, you must reimplement this function. An incomplete prototype implementation is in `rt_memory.s`.

Implementation

The calling convention is ordinary ATPCS. On entry, r0 is the minimum size of the desired block, and r1 holds a pointer to a location to store the base address in.

Returns

The default implementation calls `__user_heap_extend()` and returns a failure indication if that function fails. On exit, r0 is the size of the block acquired, or 0 if nothing could be obtained, and the memory location r1 pointed to on entry will contain the base address of the block.

4.9.11 __rt_stack_postlongjmp()

This function sets sp and sl to correct values after a call to longjmp. An incomplete prototype implementation in assembler code is in `rt_memory.s`

Implementation

This function is called with r0 containing the pre-setjmp value for sl, and r1 containing the pre-setjmp value for sp.

If your memory model is used only with non-stack-checked ATPCS, you do not have to implement this function.

Returns

The function must set sl and sp to valid post-longjmp values. The registers must be set atomically in order to avoid interrupt problems. So in the minimal implementation where the memory model requires no special handling, you would push r0 and r1 on the stack and then use `LDM` to load sl and sp atomically with the new values.

4.10 Tailoring the input/output functions

The higher-level input/output functions such as `fscanf()` and `fprintf()` are not target dependent. However, the higher-level functions perform input/output by calling lower-level functions that are target dependent. To retarget input/output, you can either avoid these higher-level functions or redefine the lower-level functions.

See the `rt_sys.h` include file for more information on I/O functions.

4.10.1 Dependencies on low-level functions

The dependencies of the higher-level function on lower-level functions is shown in Table 4-12. If you define your own versions of the lower-level functions, you can use the library versions of the higher-level functions directly. `fgetc()` uses `__FILE`, but `fputc()` uses `__FILE` and `ferror()`.

Table 4-12 Input/Output Dependencies

Low-level object	Description	fprintf	printf	fwrite	fputs	puts	fscanf	scanf	fread	read	fgets	gets
`__FILE`	The file structure	*	*	*	*	*	*	*	*	*	*	*
`__stdin`	The standard input object of type `__FILE`	-	-	-	-	-	-	*	-	*	-	*
`__stdout`	The standard output object of type `__FILE`	-	*	-	-	*	-	-	-	-	-	-
`fputc()`	Outputs a character to a file	*	*	*	*	*	-	-	-	-	-	-
`ferror()`	Returns the error status accumulated during file input/output	*	*	*	-	-	-	-	-	-	-	-
`fgetc()`	Gets a character from a file	-	-	-	-	-	*	*	*	*	*	*
`__backspace()`	Moves file pointer to previous character	-	-	-	-	-	*	*	-	-	-	-

Refer to the ANSI C Reference for syntax of the low-level functions.

printf family

The printf family consists of _printf(), printf(), _fprintf(), fprintf(), vprintf(), and vfprintf(). All these functions use __FILE opaquely and depend only on the functions fputc and ferror. The functions _printf() and _fprintf() are identical to printf() and fprintf() except that they cannot format floating-point values.

The standard output functions of the form _printf(...) are equivalent to:

```
fprintf(& __stdout, ...)
```

where __stdout has type __FILE.

scanf family

The scanf family consists of scanf() and fscanf(). These functions depend only on the functions fgetc(), __FILE, and __backspace().

The standard input form scanf(...) is equivalent to:

```
fscanf(& __stdin, ...)
```

where __stdout has type __FILE.

fwrite(), fputs, and puts

If you define your own version of __FILE, and your own fputc() and ferror() functions and the __stdout object, you can use all of the printf family, fwrite(), fputs(), and puts() unchanged from the library. Example 4-19 shows how to do this.

Example 4-19

```
#include <stdio.h>
struct __FILE {
    int handle;
    /* Whatever you need here */
};
FILE __stdout;
int fputc(int ch, FILE *f)
{
    /* Your implementation of fputc */
    return ch;
}
int ferror(FILE *f)
{
    /* Your implementation of ferror */
    return EOF;
}
void test(void)
{
    printf("Hello world\n");  /* This works ... */
}
```

By default, `fread()` and `fwrite()` call fast block input/output functions that are part of the ARM stream implementation. If you define your own `__FILE` structure instead of using the ARM stream implementation, `fread()` and `fwrite()` will call `fgetc()` instead of calling the block input/output functions.

fread(), fgets(), and gets()

The functions `fread()`, `fgets()`, and `gets()` are implemented as a loop over `fgetc()` and `ferror()`. Each uses the `FILE` argument opaquely.

If you provide your own implementation of `__FILE`, `__stdin` (for `gets()`), `fgetc()`, and `ferror()`, you can use these functions directly from the library.

4.10.2 Target-dependent input/output support functions

`rt_sys.h` defines the type `FILEHANDLE`. The value of `FILEHANDLE` is returned by `_sys_open()` and identifies an open file on the host system.

The target-dependent input and output functions and their library members are listed in Table 4-13.

Table 4-13 I/O support functions

Function	Description
`_sys_open()`	*_sys_open()* on page 4-70
`_sys_close()`	*_sys_close()* on page 4-71
`_sys_read()`	*_sys_seek()* on page 4-74
`_sys_write()`	*_sys_write()* on page 4-72
`_sys_seek()`	*_sys_read()* on page 4-71
`_sys_ensure()`	*_sys_ensure()* on page 4-73
`_sys_flen()`	*_sys_flen()* on page 4-73
`_sys_istty()`	*_sys_istty()* on page 4-74
`_sys_tmpnam()`	*_sys_tmpnam()* on page 4-75
`_sys_command_string()`	*_sys_command_string()* on page 4-75

The default implementation of these functions is semihosted. That is, each function uses the semihosting SWI. If any function is redefined, all stream-support functions must be redefined.

4.10.3 _sys_open()

This function opens a file.

Syntax

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

Implementation

The `_sys_open` function is required by `fopen()` and `freopen()`. These functions, in turn, are required if any file input/output function is to be used.

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

Returns

The return value is –1 if an error occurs.

4.10.4 _sys_close()

This function closes a file previously opened with `_sys_open()`.

Syntax

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

Implementation

This function must be defined if any input/output function is to be used.

Returns

The return value is 0 if successful. A non-zero value indicates an error.

4.10.5 _sys_read()

This function reads the contents of a file into a buffer.

Syntax

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

Implementation

The *`mode`* argument is a bitmap describing the state of the file connected to *`fh`*, as for `_sys_write()`.

Returns

The return value is one of the following:

- the number of characters *not* read (that is, *`len`* - *`result`* were read)
- an error indication
- an EOF indicator. The EOF indication involves the setting of `0x80000000` in the normal result. The target-independent code is capable of handling either:

early EOF	The last read from a file returns some characters plus an EOF indicator
late EOF	The last read returns just EOF.

4.10.6 _sys_write()

Writes the contents of a buffer to a file previously opened with `_sys_open()`.

Syntax

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

Implementation

The *`mode`* parameter is a bitmap describing the state of the file connected to *`fh`*, whether it is a binary file, and how it is buffered. The mode bits might be important if the file is connected to a terminal device because they specify whether or not the device is to be used raw (for example, whether the terminal input should be echoed). See the `_IOxxx` constants in `stdio.h` for definitions of user-accessible mode bits.

Returns

The return value is either:

- a positive number representing the number of characters *not* written (so any non-zero return value denotes a failure of some sort)
- a negative number indicating an error.

4.10.7 _sys_ensure()

This function flushes buffers associated with a file handle.

Syntax

```
int _sys_ensure(FILEHANDLE fh)
```

Implementation

A call to `_sys_ensure()` flushes any buffers associated with file handle *fh*, and ensures that the file is up to date on the backing store medium.

Returns

If an error occurs, the result is negative.

4.10.8 _sys_flen()

This function returns the current length of a file.

Syntax

```
long _sys_flen(FILEHANDLE fh)
```

Implementation

The function is required in order to convert `fseek(, SEEK_END)` into `(, SEEK_SET)` as required by `_sys_seek()`.

If `fseek()` is used with an underlying system that does not directly support seeking relative to the end of a file, `_sys_flen()` must be defined. If the underlying system can seek relative to the end of a file, you can define `fseek()` such that `_sys_flen()` is not required.

Returns

This function returns the current length of the file *fh*, or a negative error indicator.

4.10.9 _sys_seek()

This function puts the file pointer at offset *pos* from the beginning of the file.

Syntax

```
int _sys_seek (FILEHANDLE fh, long pos)
```

Implementation

This function sets the current read or write position to the new location *pos* relative to the start of the current file *fh*.

Returns

The result is non-negative if no error occurs or is negative if an error occurs.

4.10.10 _sys_istty()

This function determines if a file handle identifies a terminal.

Syntax

```
int _sys_istty(FILE *f)
```

Implementation

When a file is connected to a terminal device, this function is used to provide unbuffered by default behavior (in the absence of a call to `set(v)buf`) and to disallow seeking.

Returns

The return value is:

0	There is not an interactive device
1	There is an interactive device
other	An error occurred.

4.10.11 _sys_tmpnam()

This function converts the file number *fileno* for a temporary file to a unique filename, for example `tmp0001`.

Syntax

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

Implementation

The function must be defined if `tmpnam()` or `tmpfile()` is used.

Returns

Returns the filename in *`name`*.

4.10.12 _sys_command_string()

This function retrieves the command line used to invoke the current application from the environment that called the application.

Syntax

```
char *_sys_command_string(char *cmd, int len)
```

where:

`cmd`	is a pointer to a buffer that can be used to store the command line. It is not required that the command line is stored in *`cmd`*.
`len`	is the length of the buffer.

Implementation

This function is called by the library startup code to set up `argv` and `argc` to pass to `main()`.

Returns

The function must return either:

- A pointer to the command line, if successful. This can be either a pointer to the *`cmd`* buffer if it is used, or a pointer to wherever else the command line is stored.
- `NULL`, if not successful.

4.11 Tailoring other C library functions

Implementation of the following ANSI standard functions depends entirely on the target operating system. None of the functions listed below is used internally by the library. So if any of these functions are not implemented, only applications calling the function directly will fail.

The target-dependent ANSI C library functions are listed in Table 4-14.

Table 4-14 ANSI C library functions

Function	Description
`clock` and `_clock_init()`	*clock()* on page 4-77 and *_clock_init()* on page 4-77
`time()`	*time()* on page 4-77
`remove()`	*remove()* on page 4-78
`rename()`	*rename()* on page 4-78
`system()`	*system()* on page 4-79
`getenv()` and `_getenv_init()`	*getenv()* on page 4-79 and *_getenv_init()* on page 4-80

The default implementation of these functions is semihosted. That is, each function uses the semihosting SWI.

`clock()` and `_clock_init()` must be reimplemented together or not at all.

4.11.1 clock()

This is the standard C library clock function from `time.h`.

Syntax

```
clock_t clock(void)
```

Implementation

If the units of `clock_t` differ from the default of centiseconds you must define `__CLK_TCK` on the compiler command line or in your own header file. The value in the definition will be used for `CLK_TCK` and `CLOCKS_PER_SEC`. If you re-implement `clock()` you must also re-implement `_clock_init()`.

Returns

The returned value is an unsigned integer.

4.11.2 _clock_init()

This is an optional initialization function for `clock()`.

Syntax

```
__weak void _clock_init(void)
```

Implementation

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

4.11.3 time()

This is the standard C library `time()` function from `time.h`.

Syntax

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

The return values is an approximation of the current calendar time.

Returns

The value `(time_t*)-1` is returned if the calendar time is not available. If timer is not a `NULL` pointer, the return value is also assigned to the `time_t*`.

4.11.4 remove()

This is the standard C library `remove()` function from `stdio.h`.

Syntax

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

Implementation

`remove()` causes the file whose name is the string pointed to by *filename* to be removed. Subsequent attempts to open the file will fail, unless it is created again. If the file is open, the behavior of the remove function is implementation-defined.

Returns

Returns zero if the operation succeeds or nonzero if it fails.

4.11.5 rename()

This is the standard C library `rename()` function from `stdio.h`.

Syntax

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

Implementation

`rename()` causes the file whose name is the string pointed to by *old* to be henceforth known by the name given by the string pointed to by *new*. The file named *old* is effectively removed. If a file named by the string pointed to by *new* exists prior to the call of the rename function, the behavior is implementation-defined.

Returns

Returns zero if the operation succeeds or nonzero if it fails. If nonzero and the file existed previously it is still known by its original name.

4.11.6 system()

This is the standard C library `system()` function from `stdlib.h`.

Syntax

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

Implementation

`system()` passes the string pointed to by *string* to the host environment to be executed by a command processor in an implementation-defined manner. A null pointer may be used for *string*, to inquire whether a command processor exists.

Returns

If the argument is a null pointer, the system function returns non-zero only if a command processor is available.

If the argument is not a null pointer, the system function returns an implementation-defined value.

4.11.7 getenv()

This is the standard C library `getenv()` function from `stdlib.h`.

Syntax

```
char *getenv(const char *name)
```

Implementation

The default implementation returns `NULL` indicating that no environment information is available. You can re-implement `getenv()` yourself. It depends on no other function and no other function depends on it.

If you redefine the function, you can also call a function `_getenv_init()` which the C library initialization code will call when the library is initialized, that is before `main()` is entered.

The function searches the environment list, provided by the host environment, for a string that matches the string pointed to by *name*. The set of environment names and the method for altering the environment list are implementation-defined.

Returns

The return value is a pointer to a string associated with the matched list member. The array pointed to must not be modified by the program, but might be overwritten by a subsequent call to `getenv()`.

4.11.8 _getenv_init()

This allows a user version of `getenv()` to initialize itself.

Syntax

```
void _getenv_init(void)
```

Implementation

If this function is defined, the C library initialization code will call when the library is initialized, that is before `main()` is entered.

4.12 ISO implementation definition

This section describes how the libraries fulfill the requirements of the ANSI specification.

4.12.1 ANSI C library implementation definition

The ANSI C library variants are listed in *Library naming conventions* on page 4-96.

The ANSI specification leaves some details to the implementors, but requires their implementation choices to be documented. The implementation details are described in this section.

- The macro `NULL` expands to the integer constant 0.
- If a program redefines a reserved external identifier, an error might occur when the program is linked with the standard libraries. If it is not linked with standard libraries, no error will be diagnosed.
- The `assert()` function prints the following message and then calls the `abort()` function as shown in Example 4-20.

Example 4-20

```
*** assertion failed: expression, file _FILE_, line _LINE_
```

The following functions test for character values in the range EOF (–1) to 255 (inclusive):

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

Mathematical functions

The mathematical functions shown in Table 4-15, when supplied with out-of-range arguments respond in the way shown.

Table 4-15 Mathematical functions

Function	Condition	Returned value	Error number
`acos(x)`	`abs(x) > 1`	`QNaN`	`EDOM`
`asin(x)`	`abs(x) > 1`	`QNaN`	`EDOM`
`atan2(x,y)`	`x =0, y = 0`	`QNaN`	`EDOM`
`atan2(x,y)`	`x = Inf, y = Inf`	`QNaN`	`EDOM`
`cos(x)`	`x=Inf`	`QNaN`	`EDOM`
`cosh(x)`	Overflow	`+Inf`	`ERANGE`
`exp(x)`	Overflow	`+Inf`	`ERANGE`
`exp(x)`	Underflow	`+0`	`ERANGE`
`fmod(x,y)`	`x=Inf`	`QNaN`	`EDOM`
`fmod(x,y)`	`y = 0`	`QNaN`	`EDOM`
`log(x)`	`x < 0`	`QNaN`	`EDOM`
`log(x)`	`x = 0`	`-Inf`	`EDOM`
`log10(x)`	`x < 0`	`QNaN`	`EDOM`
`log10(x)`	`x = 0`	`-Inf`	`EDOM`
`pow(x,y)`	Overflow	`+Inf`	`ERANGE`
`pow(x,y)`	Underflow	`0`	`ERANGE`
`pow(x,y)`	`x=0 or x=Inf, y=0`	`+1`	`EDOM`
`pow(x,y)`	`x=+0, y<0`	`-Inf`	`EDOM`
`pow(x,y)`	`x=-0,` `y<0 and y integer`	`-Inf`	`EDOM`
`pow(x,y)`	`x= -0,` `y<0 and y noninteger`	`QNaN`	`EDOM`
`pow(x,y)`	`x<0, y noninteger`	`QNaN`	`EDOM`

Table 4-15 Mathematical functions (Continued)

Function	Condition	Returned value	Error number
`pow(x,y)`	`x=1, y=Inf`	`QNaN`	`EDOM`
`sqrt(x)`	`x < 0`	`QNaN`	`EDOM`
`sin(x)`	`x=Inf`	`QNaN`	`EDOM`
`sinh(x)`	Overflow	`+Inf`	`ERANGE`
`tan(x)`	`x=Inf`	`QNaN`	`EDOM`
`atan(x)`	`SNaN`	`SNaN`	None
`ceil(x)`	`SNaN`	`SNaN`	None
`floor(x)`	`SNaN`	`SNaN`	None
`frexp(x)`	`SNaN`	`SNaN`	None
`ldexp(x)`	`SNaN`	`SNaN`	None
`modf(x)`	`SNaN`	`SNaN`	None
`tanh(x)`	`SNaN`	`SNaN`	None

`HUGE_VAL` is an alias for `Inf`. Consult the `errno` variable for the error number. Other than the cases shown in Table 4-15, all functions return `QNaN` when passed `QNaN` and throw an invalid operation exception when passed `SNaN`.

Signal function

The signals listed in Table 4-16 are supported by the `signal()` function

Table 4-16 Signal function signals

Signal	Number	Description	Additional argument
SIGABRT	1	Abort	None
SIGFPE	2	Arithmetic exception	A set of bits from {FE_EX_INEXACT, FE_EX_UNDERFLOW, FE_EX_OVERFLOW, FE_EX_DIVBYZERO, FE_EX_INVALID, DIVBYZERO}
SIGILL	3	Illegal instruction	None
SIGINT	4	Attention request from user	None
SIGSEGV	5	Bad memory access	None
SIGTERM	6	Termination request	None
SIGSTAK	7	Stack overflow	None
SIGRTRED	8	Redirection failed on a runtime library input/output stream	Name of file or device being re-opened to redirect a standard stream
SIGRTMEM	9	Out of heap space	Size of failed request
SIGUSR1	10	User-defined	User-defined
SIGUSR2	11	User-defined	User-defined
reserved	12 - 31	Reserved	Reserved
other	> 31	User-defined	User-defined

A signal number greater than 31 can be passed through `__raise()`, and caught by the default signal handler, but it cannot be caught by a handler registered using `signal()`.

`signal()` returns an error code if you try to register a handler for a signal number greater than `SIGUSR2`.

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

Caution

The IEEE 754 standard for floating-point processing states that the default action to an exception is to proceed without a trap. A raised exception in floating-point calculations does not, by default, generate SIGFPE. You can modify fp error handling by tailoring the functions and definitions in `fenv.h`. See *Tailoring error signalling, error handling, and program exit* on page 4-47 and the chapter on floating-point in the *ADS Developer Guide*.

For all the signals in Table 4-16, when a signal occurs, if the handler points to a function, the equivalent of `signal(sig, SIG_DFL)` is executed before the call to handler.

If the **SIGILL** signal is received by a handler specified to by the `signal()` function, the default handling is reset.

Input/output characteristics

The generic ARM C library has the following input/output characteristics:

- The last line of a text stream does not require a terminating newline character.
- Space characters written out to a text stream immediately before a newline character do appear when read back in.
- No null characters are appended to a binary output stream.
- The file position indicator of an append mode stream is initially placed at the end of the file.
- A write to a text stream causes the associated file to be truncated beyond the point where the write occurred if this is the behavior of the device category of the file.
- The characteristics of file buffering agree with section 4.9.3 of the ANSI C standard. If semihosting is used, the maximum number of open files is limited by the available target memory.
- A zero-length file, into which no characters have been written by an output stream, does exist.
- A file can be opened many times for reading, but only once for writing or updating. A file cannot simultaneously be open for reading on one stream, and open for writing or updating on another.
- Local time zones and Daylight Saving Time are not implemented. The values returned will indicate that the information is not available. For example, the `gmtime()` function always returns NULL.

- 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 semihosting SWI, however, does not pass the status back to the execution environment.
- The error messages returned by the `strerror()` function are identical to those given by the `perror()` function.
- If the size of area requested is zero, `calloc()`, `malloc()` and `realloc()` return `NULL`.
- `abort()` closes all open files and deletes all temporary files.
- `fprintf()` prints `%p` arguments in lowercase hexadecimal format 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 exactly the same as `%x` arguments.
- `fscanf()` always treats the character "-" in a `%...[...]` argument as a literal character.
- `ftell()` and `fgetpos()` set `errno` to the value of EDOM on failure.
- `perror()` generates the messages in Table 4-17.

Table 4-17 perror() messages

Error	Message
0	`No error (errno = 0)`
EDOM	`EDOM - function argument out of range`
ERANGE	`ERANGE - function result not representable`
ESIGNUM	`ESIGNUM - illegal signal number`
Others	`Unknown error`

The following characteristics, required to be specified in an ANSI-compliant implementation, are unspecified in the 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()`.

4.12.2 Standard C++ library implementation definition

This section describes the implementation of the C++ libraries. The ARM C++ library provides all of the library defined in the *ISO/IEC 14822 :1998 International Standard for C++*, aside from some limitations described below. For information on implementation-defined behavior that is defined in the Rogue Wave C++ library, refer to the included Rogue Wave HTML documentation. By default, this is installed in the *`install_directory`*`\HTML`.

The standard C++ library is distributed in binary form only.

The requirements that the C++ library places on the C library are described in Table 4-18.

Table 4-18 C++ requirements on the C library

File	Required function in C library
ctype.h	`isalnum, isalpha, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit, tolower, toupper`
locale.h	`localeconv, setlocale`
math.h	`acos, asin, atan2, atan, ceil, cos, cosh, exp, fabs, floor, fmod, frexp, ldexp, log10, log, modf, pow, sin, sinh, sqrt, tan, tanh`
setjmp.h	`longjmp`
signal.h	`raise, signal`
stdio.h	`clearerr, fclose, feof, ferror, fflush, fgetc, fgetpos, fgets, fopen, fprintf, fputc, fputs, fread, freopen, fscanf, fseek, fsetpos, ftell, fwrite, getc, getchar, gets, perror, printf, putc, putchar, puts, remove, rename, rewind, scanf, setbuf, setvbuf, sprintf, sscanf, tmpfile, tmpnam, ungetc, vfprintf, vprintf, vsprintf`
stdlib.h	`abort, abs, atexit, atof, atoi, atol, bsearch, calloc, div, exit, free, getenv, labs, ldiv, malloc, mblen, qsort, rand, realloc, srand, strtod, strtol, strtoul, system`
string.h	`memchr, memcmp, memcpy, memmove, memset, memset, strcat, strchr, strcmp, strcoll, strcpy, strcspn, strerror, strlen, strncat, strncmp, strncpy, strpbrk, strrchr, strspn, strstr, strtok, strxfrm`
time.h	`asctime, clock, ctime, difftime, mktime, strftime, time`

The most important features missing from this release are described in Table 4-19.

Table 4-19 Standard C++ library differences

Standard	Implementation differences
Wide character	Not a separate type. `wchar_t` is an existing typedef for **`int`**. Characters are 8-bits wide.
Namespaces	Not supported. All top-level items are in the global namespace.
Unimplemented features	Support functions for unimplemented language features, `class bad_cast` for example, are unlikely to be functional.
locale	The locale message facet is not supported. It will fail to open catalogs at runtime because the ARM C library does not support `catopen` and `catclose` through `nl_types.h`. One of two locale definitions can be selected at link time, other locales can be created by user-redefinable functions.
Timezone	Not supported. The ARM C library does not support it.
Complex default template arguments	Not supported. Complex default template argument definitions are where a type parameter has a default instantiation involving an earlier type parameter. When you request a template that the standard says is defined with a complex default (such as instantiating class queue), you must always supply a value for each template parameter. No defaults will be present.
Exceptions	Not supported.
`typeinfo`	Limited support. **`typeinfo`** is supported in a basic way by the ARM C++ library additions.

-

4.13 C library extensions

This section describes the ARM-specific library extensions and functions defined by the C9X draft standard.

Table 4-20 Extensions

Function	Header file definition	Entension
atoll() on page 4-90	`stdlib.h`	C9X draft standard
strtoll() on page 4-91	`stdlib.h`	C9X draft standard
strtoull() on page 4-91	`stdlib.h`	C9X draft standard
snprintf() on page 4-91	`stdio.h`	C9X draft standard
vsnprintf() on page 4-92	`stdio.h`	C9X draft standard
lldiv() on page 4-92	`stdlib.h`	C9X draft standard
llabs() on page 4-93	`stdlib.h`	C9X draft standard
alloca() on page 4-93	`alloca.h`	C9X and others
_fisatty() on page 4-93	`stdio.h`	ARM-specific
__heapstats() on page 4-94	`stdlib.h`	ARM-specific
__heapvalid() on page 4-94	`stdlib.h`	ARM-specific

4.13.1 atoll()

The `atoll()` function converts a decimal string into an integer, similarly to the ANSI functions `atol()` and `atoi()`, but returning a **`long long`** result. Like `atoi()`, `atoll()` can accept octal or hexadecimal input if the string begins with `0` or `0x`.

Syntax

```
long long atoll(const char *nptr)
```

4.13.2 strtoll()

The `strtoll()` function converts a string in an arbitrary base to an integer, similarly to the ANSI function `strtol()`, but returning a **`long long`** result. Like `strtol()`, the parameter *`endptr`* can point to a location in which to store a pointer to the end of the translated string, or can be `NULL`. The parameter *`base`* should contain the number base. Setting *`base`* to zero indicates that the base should be selected in the same way as `atoll()`.

Syntax

```
long long strtoll(const char *nptr, char **endptr, int base)
```

4.13.3 strtoull()

`strtoull()` is exactly the same as `strtoll()`, but returns an **`unsigned long long`**.

Syntax

```
unsigned long long strtoull(const char *nptr, char **endptr, int
base)
```

4.13.4 snprintf()

`snprintf()` works almost exactly like the ANSI `sprintf()` function, except that the caller can specify the maximum size of the buffer. The return value is the length of the complete formatted string that would have been written if the buffer were big enough. Therefore, the string written into the buffer is complete only if the return value is at least zero and at most `n-1`.

The *`bufsize`* parameter specifies the number of characters of *`buffer`* that the function can write into, *including* the terminating null.

`<stdio.h>` is an ANSI header file, but the function is not allowed by the ANSI C library standard. it therefore not available if you use the compilers with the `-strict` option.

Syntax

```
int snprintf(char *buffer, size_t bufsize, const  char *format,
...)
```

4.13.5 vsnprintf()

`vsnprintf()` works almost exactly like the ANSI `vsprintf()` function, except that the caller can specify the maximum size of the buffer. The return value is the length of the complete formatted string that would have been written if the buffer were big enough. Therefore, the string written into the buffer is complete only if the return value is at least zero and at most `n-1`.

The *`bufsize`* parameter specifies the number of characters of *`buffer`* that the function can write into, *including* the terminating null.

`<stdio.h>` is an ANSI header file, but the function is not allowed by the ANSI C library standard. it therefore not available if you use the compilers with the `-strict` option.

Syntax

```
int vsnprintf(char *buffer,  size_t bufsize, const  char
*format,   va_list ap)
```

4.13.6 lldiv()

The `lldiv` function divides two **`long long`** integers and returns both the quotient and the remainder. It is the **`long long`** equivalent of the ANSI function `ldiv`. The return type `lldiv_t` is a structure containing two **`long long`** members, called `quot` and `rem`.

`<stdlib.h>` is an ANSI header file, but the function is not allowed by the ANSI C library standard. it therefore not available if you use the compilers with the `-strict` option.

Syntax

```
lldiv_t lldiv(long longnum, long long denom)
```

-

4.13.7 llabs()

The `llabs` returns the absolute value of its input. It is the **`long long`** equivalent of the ANSI function `labs`.

`<stdlib.h>` is an ANSI header file, but the function is not allowed by the ANSI C library standard. it therefore not available if you use the compilers with the `-strict` option.

Syntax

```
long long llabs(long longnum)
```

4.13.8 alloca()

The `alloca` function allocates local storage in a function. It returns a pointer to *`size`* bytes of memory, or `NULL` if not enough memory was available.

Memory returned from `alloca` should never be passed to `free`. Instead, the memory will be deallocated automatically when the function that called `alloca` returns.

`alloca()` should not be called via a function pointer. Care should be taken when using `alloca` and `setjmp` in the same function, since memory allocated by `alloca()` between calling `setjmp` and `longjmp` will be deallocated by the call to `longjmp`.

This function is a common non-standard extension to many C libraries.

Syntax

```
void* alloca(size_t size)
```

4.13.9 _fisatty()

The `_fisatty` function determines whether the given `stdio` stream is attached to a terminal device or a normal file. It calls the `_sys_istty` low-level function (see *Tailoring the input/output functions* on page 4-67) on the underlying file handle. It returns 1 for a terminal, 0 for a file, and less than 0 for an error.

This function is an ARM-specific library extension.

Syntax

```
int _fisatty(FILE *stream)
```

4.13.10 __heapstats()

The `__heapstats` function displays statistics on the state of the storage allocation heap. It calls the `__Heap_Stats` function, which you can re-implement if you choose to do your own storage management (see *__Heap_Stats()* on page 4-58). The ARM default implementation gives information on how many free blocks exist in various size ranges.

The function outputs its results by calling the supplied output function *`dprint`*, which should work essentially like `fprintf()`. The first parameter passed to *`dprint`* is the supplied pointer *`param`*. You can pass `fprintf()` itself, provided you cast it to the right function pointer type. This type is defined as a **`typedef`** for convenience: it is called `__heapprt`. For example:

```
__heapstats((__heapprt)fprintf, stderr);
```

If you call `fprintf()` on a stream that you have not already sent output to, the library will call `malloc` internally to create a buffer for the stream. If this happens in the middle of a call to `__heapstats()`, the heap may be corrupted. You should therefore ensure you have already sent some output to `stderr` in the above example.

This function is an ARM-specific library extension.

Syntax

```
void __heapstats(int (*dprint)( void*param, char const
*format,...), void* param)
```

4.13.11 __heapvalid()

The `__heapvalid` function performs a consistency check on the heap. It outputs detailed information about every free block if the *`verbose`* parameter is non-zero, and only output errors otherwise.

The function outputs its results by calling the supplied output function *`dprint`*, which should work essentially like `fprintf()`. The first parameter passed to *`dprint`* is the supplied pointer *`param`*. You can pass `fprintf()` itself, provided you cast it to the right function pointer type. This type is defined as a **`typedef`** for convenience: it is called `__heapprt`. For example:

Example 4-21 Calling __heapvalid with fprintf

```
__heapvalid((__heapprt) fprintf, stderr, 0);
```

If you call `fprintf()` on a stream that you have not already sent output to, the library will call `malloc()` internally to create a buffer for the stream. If this happens in the middle of a call to `__heapvalid()`, the heap may be corrupted. You should therefore ensure you have already sent some output to `stderr`. The code in Example 4-21 will cause a major failure if you have not already written to the stream.

This function is an ARM-specific library extension.

Syntax

```
void __heapvalid(int (*dprint)( void*param, char const
*format,...), void* param, intverbose)
```

4.14 Library naming conventions

The filename identifies how the variant was built as follows:

root_<arch><fpu><dfmt><stack><entrant>.<endian>

The values for the fields of the name are listed below:

root	`c`	ANSI C and basic runtime support
	`f`	C/Java rounding and exception options for fp arithmetic
	`g`	Full IEEE rounding and exception options for fp arithmetic
	`m`	Transcendental math functions
	`cpp`	High-level C++ functions
	`cpprt`	Low-level and runtime support C++ functions.
arch	`a`	An ARM library
	`t`	A Thumb library.
fpu	`f`	Uses FPA instruction set
	`v`	Uses VFP instruction set
	`_`	Does not use floating-point instructions.
dfmt	`p`	Pure-endian double format
	`m`	Mixed-endian double format
	`_`	No use of floating-point doubles.
stack	`u`	Does not use stack checking
	`s`	Uses software stack checking
	`_`	Not applicable.
entrant	`n`	The functions are not reentrant
	`e`	The functions are reentrant
	`_`	Not applicable.
endian	`l`	Little-endian
	`b`	Big-endian
	`_`	Not applicable.

The 8 C library names are `c_{a,t}__{s,u}{e,n}`

c_a__se	ARM, stack checking, reentrant
c_a__sn	ARM, stack checking, not reentrant
c_a__ue	ARM, no stack checking, reentrant
c_a__un	ARM, no stack checking, not reentrant (base PCS)

c_t__se	Thumb, stack checking, reentrant
c_t__sn	Thumb, stack checking, not reentrant
c_t__ue	Thumb, no stack checking, reentrant
c_t__un	Thumb, no stack checking, not reentrant (base PCS).

The 10 FPLIB names are `f_{a,t}[fm, vp, _m, _p]`

f_afm	ARM, FPA, mixed-endian double
f_avp	ARM, VFP, pure-endian double
f_a_m	ARM, soft FPA
f_a_p	ARM, soft VFP
f_a	ARM, used with `-fpu none`
f_tfm	Thumb, FPA, mixed-endian double
f_tvp	Thumb,VFP, pure-endian double
f_t_m	Thumb, mixed-endian double
f_t_p	Thumb, pure-endian double
f_t	Thumb, used with `-fpu none.`

The 16 MATHLIB names are `m_{a,t}{fm, vp, _m, _p}{s,u}`

m_afms	ARM, FPA, mixed-endian, stack checking
m_afmu	ARM, FPA, mixed-endian, no stack checking
m_avps	ARM, VFP, pure-endian, stack checking
m_avpu	ARM, VFP, pure-endian, no stack checking
m_a_ms	ARM, mixed-endian, stack checking
m_a_mu	ARM, mixed-endian, no stack checking
m_a_ps	ARM, pure-endian, stack checking
m_a_pu	ARM, pure-endian, no stack checking
m_tfms	Thumb, FPA, mixed-endian, stack checking
m_tfmu	Thumb, FPA, mixed-endian, no stack checking
m_tvps	Thumb, VFP, pure-endian, stack checking
m_tvpu	Thumb, VFP, pure-endian, no stack checking
m_t_ms	Thumb, mixed-endian, stack checking
m_t_mu	Thumb, mixed-endian, no stack checking
m_t_ps	Thumb, pure-endian, stack checking
m_t_pu	Thumb, pure-endian, no stack checking.

See *Specifying the target processor or architecture* on page 2-17 for details on selecting a specific architecture or processor selection.

Chapter 5
Assembler

This chapter describes the language features that are provided by the ARM assembler, such as pseudo-instructions, directives and macros. It does not contain information about the inline assemblers in the ARM C and C++ compilers (see the Mixed Language Programming chapter in *ADS Developer Guide*). It contains the following sections:

See Table 5-1 on page 5-2 to Table 5-7 on page 5-3 to locate individual directives or pseudo-instructions.

-

5.1 Introduction

This chapter does not contain detailed information on how to write ARM assembly language. See the assembly language chapter in the *ADS Developer Guide* for tutorial information.

For detailed information on ARM and Thumb instruction mnemonics, see *ARM Architecture Reference Manual*.

5.1.1 Location of directives and pseudo-instructions

Table 5-1 Directives, symbol definition

`CN` on page 5-42	`GBLA` on page 5-79	`LCLS` on page 5-90
`CP` on page 5-45	`GBLL` on page 5-80	`RLIST` on page 5-100
`DN` on page 5-60	`GBLS` on page 5-81	`RN` on page 5-101
`EQU or *` on page 5-64	`IMPORT` on page 5-84	`SETA` on page 5-103
`EXPORT or GLOBAL` on page 5-65	`KEEP` on page 5-87	`SETL` on page 5-104
`EXTERN` on page 5-66	`LCLA` on page 5-88	`SETS` on page 5-105
`FN` on page 5-69	`LCLL` on page 5-89	`SN` on page 5-106

Table 5-2 Directives, data definition

`ALIGN` on page 5-37	`DCFDU` on page 5-52	`DCWU` on page 5-59
`DATA` on page 5-46	`DCFS` on page 5-53	`FIELD or #` on page 5-68
`DCB or =` on page 5-47	`DCFSU` on page 5-54	`LTORG` on page 5-91
`DCD or &` on page 5-48	`DCI` on page 5-55	`MAP or ^` on page 5-95
`DCDO` on page 5-49	`DCQ` on page 5-56	`SPACE or %` on page 5-107
`DCDU` on page 5-50	`DCQU` on page 5-57	
`DCFD` on page 5-51	`DCW` on page 5-58	

Table 5-3 Directives, assembly control

`ELSE or |` on page 5-61	`INCBIN` on page 5-85	`WEND` on page 5-112
`ENDIF or ]` on page 5-62	`MACRO` on page 5-92	`WHILE` on page 5-113
`GET or INCLUDE` on page 5-82	`MEND` on page 5-96	
`IF or [` on page 5-83	`MEXIT` on page 5-96	

Table 5-4 Directives, structure

`ENDFUNC` on page 5-62	`FRAME PUSH` on page 5-72	`FRAME STATE REMEMBER` on page 5-76
`ENDP` on page 5-62	`FRAME REGISTER` on page 5-73	`FRAME STATE RESTORE` on page 5-77
`FRAME ADDRESS` on page 5-70	`FRAME RESTORE` on page 5-74	`FUNCTION` on page 5-78
`FRAME POP` on page 5-71	`FRAME SAVE` on page 5-75	`PROC` on page 5-99

Table 5-5 Directives, miscellaneous

`AREA` on page 5-39	`ENTRY` on page 5-63	`ROUT` on page 5-102
`ASSERT` on page 5-41	`INFO or !` on page 5-86	`SUBT` on page 5-108
`CODE16` on page 5-43	`NOFP` on page 5-97	`TTL` on page 5-109
`CODE32` on page 5-44	`OPT` on page 5-98	`VFPASSERT SCALAR` on page 5-110
`END` on page 5-61	`REQUIRE` on page 5-99	`VFPASSERT VECTOR` on page 5-111

Table 5-6 Directives, symbolic synonyms

`DCB or =` on page 5-47	`EQU or *` on page 5-64	`INFO or !` on page 5-86
`DCD or &` on page 5-48	`FIELD or #` on page 5-68	`MAP or ^` on page 5-95
`ELSE or |` on page 5-61	`IF or [` on page 5-83	`SPACE or %` on page 5-107
`ENDIF or ]` on page 5-62		

Table 5-7 Pseudo-instructions

ARM pseudo-instructions	Thumb pseudo-instructions
`ADR` on page 5-14	`ADR` on page 5-25
`ADRL` on page 5-15	`LDR` on page 5-26
`FLDD` on page 5-17	`MOV` on page 5-28
`FLDS` on page 5-18	`NOP` on page 5-29
`LDFD` on page 5-19	
`LDFS` on page 5-20	
`LDR` on page 5-21	
`NOP` on page 5-23	

5.2 Command syntax

Invoke the ARM assembler using this command:

```
armasm [-16|-32] [-apcs [none|[/qualifier[/qualifier[...]]]]]
[-bigend|-littleend] [-checkreglist] [-cpu cpu]
[-depend dependfile|-m|-md] [-errors errorfile] [-fpu fparch]
[-g] [-help] [-i dir [,dir]…] [-keep] [-list [listingfile]
[options]] [-maxcache n] [-memaccess attributes] [-nocache]
[-noesc] [-noregs] [-nowarn] [-o filename]
[-predefine "directive"] [-unsafe] [-via file] inputfile
```

where:

`-16` instructs the assembler to interpret instructions as Thumb instructions. This is equivalent to a `CODE16` directive at the head of the source file.

`-32` instructs the assembler to interpret instructions as ARM instructions. This is the default.

`-apcs [none|[/qualifier[/qualifier[...]]]]`

specifies whether you are using the *ARM/Thumb Procedure Call Standard* (ATPCS). It may also specify some attributes of code sections. See *ADS Developer Guide* for more information about the ATPCS.

`/none` specifies that `inputfile` does not use ATPCS. ATPCS registers are not set up. Qualifiers are not allowed.

Note

ATPCS qualifiers do not affect the code produced by the assembler. They are an assertion by the programmer that the code in `inputfile` complies with a particular variant of ATPCS. They cause attributes to be set in the object file produced by the assembler. The linker uses these attributes to check compatibility of files, and to select appropriate library variants.

Values for `qualifier` are:

`/interwork`

specifies that the code in `inputfile` is suitable for ARM/Thumb interworking. See *ADS Developer Guide* for information on interworking.

`/nointerwork`

specifies that the code in `inputfile` is not suitable for ARM/Thumb interworking. This is the default.

`/ropi` specifies that the content of *`inputfile`* is read-only position-independent. The default is `/noropi`.

`/pic` is a synonym for `/ropi`.

`/nopic` is a synonym for `/noropi`.

`/rwpi` specifies that the content of *`inputfile`* is read-write position-independent. The default is `/norwpi`.

`/pid` is a synonym for `/rwpi`.

`/nopid` is a synonym for `/norwpi`.

`/swstackcheck`

specifies that the code in *`inputfile`* carries out software stack-limit checking.

`/noswstackcheck`

specifies that the code in *`inputfile`* does not carry out software stack-limit checking. This is the default.

`/swstna` specifies that the code in *`inputfile`* is compatible both with code which carries out stack-limit checking, and with code that does not carry out stack-limit checking.

`-bigend` instructs the assembler to assemble code suitable for a big-endian ARM. The default is `-littleend`.

`-littleend`

instructs the assembler to assemble code suitable for a little-endian ARM. This is the default.

`-checkreglist`

instructs the assembler to check `RLIST`, `LDM`, and `STM` register lists to ensure that all registers are provided in increasing register number order. A warning is given if registers are not listed in order.

`-cpu` *`cpu`* sets the target cpu. Some instructions produce either errors or warnings if assembled for the wrong target cpu (see also the `-unsafe` assembler option). Valid values for *`cpu`* are:

- `3`, or `3M`
- `4`, `4T`, `4xM`, or `4TxM`
- `5T`

or part numbers such as `ARM7TDMI`. See the *ARM Architecture Reference Manual* for information about the architectures.

`-depend` *`dependfile`*

instructs the assembler to save source file dependency lists to *`dependfile`*. These are suitable for use with make utilities.

`-m` instructs the assembler to write source file dependency lists to `stdout`.

`-md` instructs the assembler to write source file dependency lists to *`inputfile`*`.d`.

`-errors` *`errorfile`*

instructs the assembler to output error messages to *`errorfile`*.

`-fpu` *`fparch`*

sets the target floating-point architecture. *`fparch`* can be any one of:

`softvfp` specifies that the code in *`inputfile`* uses VFP-format (pure-endian) double-precision floating-point representations, but does not use any floating-point coprocessor instructions. This is the default.

`softfpa` specifies that the code in *`inputfile`* uses FPA-format double-precision floating-point representations, but does not use any floating-point coprocessor instructions.

`vfp` selects the VFP coprocessor instruction set.

`fpa` selects the FPA coprocessor instruction set.

`none` specifies that the code in *`inputfile`* does not use any floating-point coprocessor instructions.

`-g` instructs the assembler to generate debug tables. Use the following command-line options to control the behavior of `-g`:

`-dwarf1` to select DWARF1 debug tables. This option is not recommended for C++. It will not be supported in future releases of ADS.

`-dwarf2` to select DWARF2 debug tables. This is the default and is selected if `-g` with no dwarf option is specified.

`-help` instructs the assembler to display a summary of the assembler command-line options.

`-i` *`dir`* `[,`*`dir`*`]…`

adds directories to the source file search path so that arguments to `GET`/`INCLUDE` directives do not need to be fully qualified (see *GET or INCLUDE directive* on page 5-82 and *INCBIN directive* on page 5-85).

`-keep` instructs the assembler to keep local labels in the symbol table of the object file, for use by the debugger (see *KEEP directive* on page 5-87).

`-list [`*`listingfile`*`] [`*`options`*`]`

instructs the assembler to output a detailed listing of the assembly language produced by the assembler to *`listingfile`*.. If `-` is given as *`listingfile`*, listing is sent to `stdout`. If no *`listingfile`* is given, listing is sent to *`inputfile`*`.lst`.

Use the following command-line options to control the behavior of `-list`:

`-noterse`

turns the `terse` flag off. When this option is on, lines skipped due to conditional assembly do not appear in the listing. If the `terse` option is off, these lines do appear in the listing. The default is on.

`-width` sets the listing page width. The default is 79 characters.

`-length` sets the listing page length. Length zero means an unpaged listing. The default is 66 lines.

`-xref` instructs the assembler to list cross-referencing information on symbols, including where they were defined and where they were used, both inside and outside macros. The default is off.

`-maxcache` *`n`*

sets the maximum source cache size to *`n`*. The default is 8MB.

`-memaccess` *`attributes`*

Specifies memory access attributes of the target memory system. The default is to allow aligned loads and saves of bytes, halfwords and words. *`attributes`* modify the default. They can be any one of the following:

`+L41`	Allow unaligned `LDR`s.
`-L22`	Disallow halfword loads.
`-S22`	Disallow halfword saves.
`-L22-S22`	Disallow halfword loads and saves.

`-nocache` turns off source caching. By default the assembler caches source files on the first pass and reads them from memory on the second pass.

`-noesc` instructs the assembler to ignore C-style escaped special characters, such as `\n` and `\t`.

`-noregs` instructs the assembler not to predefine register names. See *Predefined register and coprocessor names* on page 5-10 for a list of predefined register names.

`-nowarn` turns off warning messages.

`-o` *`filename`*

names the output object file. If this option is not specified, the assembler uses the second command-line argument that is not a valid command-line option as the name of the output file. If there is no such argument, the assembler creates an object filename of the form *`inputfilename`*`.o`.

`-predefine "`*`directive`*`"`

instructs the assembler to pre-execute one of the `SET` directives. You must enclose *`directive`* in quotes. See:

- *SETA directive* on page 5-103
- *SETL directive* on page 5-104
- *SETS directive* on page 5-105.

The assembler executes a corresponding `GBLL`, `GBLS`, or `GBLA` directive to define the variable before setting its value.

`-unsafe` allows assembly of a file containing instructions that are not available on the specified architecture and processor. Corresponding error messages are changed to warning messages.

`-via` *`file`* instructs the assembler to open *`file`* and read in command-line arguments to the assembler.

`inputfile` specifies the input file for the assembler. Input files must be ARM or Thumb assembly language source files.

5.3 Format of source lines

The general form of source lines in an ARM assembly language module is:

`{symbol} {instruction|directive|pseudo-instruction} {;comment}`

All three sections of the source line are optional.

Instructions cannot start in the first column. They must be preceded by white space even if there is no preceding symbol.

You may write directives in all upper case, as in this manual. Alternatively, you may write directives in all lower case. You must not write a directive in mixed upper and lower case.

You can use blank lines to make your code more readable.

`symbol` is usually a label (see *Labels* on page 5-33). In instructions and pseudo-instructions it is always a label. In some directives it is a symbol for a variable or a constant. The description of the directives makes this clear.

`symbol` must begin in the first column and cannot contain any whitespace character such as a space or a tab (see *Symbol naming rules* on page 5-30).

5.4 Predefined register and coprocessor names

All register and coprocessor names are case-sensitive.

5.4.1 Predeclared register names

The following register names are predeclared:

- `r0-r15` and `R0-R15`
- `a1-a4` (argument/result/scratch registers, synonyms for `r0-r3`)
- `v1-v8` (variable registers, `r4-r11`)
- `sb` and `SB` (static base, `r9`)
- `sl` and `SL` (stack limit, `r10`)
- `fp` and `FP` (frame pointer, `r11`)
- `ip` and `IP` (intra-procedure-call scratch register, `r12`)
- `sp` and `SP` (stack pointer, `r13`)
- `lr` and `LR` (link register, `r14`)
- `pc` and `PC` (program counter, `r15`).

5.4.2 Predeclared program status register names

The following program status register names are predeclared:

- `cpsr` and `CPSR` (current program status register)
- `spsr` and `SPSR` (saved program status register).

5.4.3 Predeclared floating-point register names

The following floating-point register names are predeclared:

- `f0-f7` and `F0-F7` (FPA registers)
- `s0-s31` and `S0-S31` (VFP single-precision registers)
- `d0-d15` and `D0-D15` (VFP double-precision registers).

5.4.4 Predeclared coprocessor names

The following coprocessor names and coprocessor register names are predeclared:

- `p0-p15` (coprocessors 0-15)
- `c0-c15` (coprocessor registers 0-15).

5.5 VFP directives and notation

You can make assertions about VFP vector lengths in your code, and have them checked by the assembler. See:

- *VFPASSERT SCALAR* on page 5-110
- *VFPASSERT VECTOR* on page 5-111.

If you use `VFPASSERT` directives, you must specify vector details in all VFP data processing instructions. The notation is described below. If you do not use `VFPASSERT` directives you must not use this notation.

In VFP data processing instructions, specify vectors of VFP registers using angle brackets:

- `s`*`n`* is a single-precision scalar register *`n`*
- `s`*`n`*`<>` is a single-precision vector whose length and stride are given by the current vector length and stride, starting at register *`n`*
- `s`*`n`*`<`*`L`*`>` is a single-precision vector of length L, stride 1, starting at register *`n`*
- `s`*`n`*`<`*`L`*`:`*`S`*`>` is a single-precision vector of length *`L`*, stride *`S`*, starting at register *`n`*
- `d`*`n`* is a double-precision scalar register *`n`*
- `d`*`n`*`<>` is a double-precision vector whose length and stride are given by the current vector length and stride, starting at register *`n`*
- `d`*`n`*`<`*`L`*`>` is a double-precision vector of length L, stride 1, starting at register *`n`*
- `d`*`n`*`<`*`L`*`:`*`S`*`>` is a double-precision vector of length *`L`*, stride *`S`*, starting at register *`n`*.

You can use this notation with names defined using the `DN` and `SN` directives (see *DN directive* on page 5-60 and *SN directive* on page 5-106).

You must not use this notation in the `DN` and `SN` directives themselves.

5.6 Built-in variables

Table 5-8 lists the built-in variables defined by the ARM assembler.

Built-in variables cannot be set using the `SETA`, `SETL`, or `SETS` directives. They can be used in expressions or conditions, for example:

```
        IF {ARCHITECTURE} = "4T"
```

Table 5-8 Built-in variables

`{PC}` or `.`	Address of current instruction.
`{VAR}` or `@`	Current value of the storage area location counter.
`{TRUE}`	Logical constant true.
`{FALSE}`	Logical constant false.
`{OPT}`	Value of the currently-set listing option. The `OPT` directive can be used to save the current listing option, force a change in it, or restore its original value.
`{CONFIG}`	Has the value `32` if the assembler is assembling ARM code, and the value `16` if it is assembling Thumb code.
`{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}`	Is a synonym for `{CONFIG}`.
`{CPU}`	Holds the name of the selected cpu. The default is `ARM7TDMI`.
`{FPU}`	Holds the name of the selected fpu. The default is `SoftVFP`.
`{ARCHITECTURE}`	Holds the name of the selected ARM architecture.
`{PCSTOREOFFSET}`	Is the offset between the address of the `STR pc,[...]` or `STM Rb,{..., pc}` instruction and the value of pc stored out. This varies depending on the CPU and architecture specified.
`{ARMASM_VERSION}`	Holds an integer that increases with each version.
`{INTER}`	Has the value True if `/inter` is set. The default is False.
`{ROPI}`	Has the value True if `/ropi` is set. The default is False.
`{RWPI}`	Has the value True if `/rwpi` is set. The default is False.
`{SWST}`	Has the value True if `/swst` is set. The default is False.
`{NOSWST}`	Has the value True if `/noswst` is set. The default is False.

5.7 ARM pseudo-instructions

The ARM assembler supports a number of pseudo-instructions that are translated into the appropriate combination of ARM or Thumb instructions at assembly time.

The pseudo-instructions available in ARM state are described in the following sections:

- *ADR ARM pseudo-instruction* on page 5-14
- *ADRL ARM pseudo-instruction* on page 5-15
- *FLDD ARM pseudo-instruction* on page 5-17
- *FLDS ARM pseudo-instruction* on page 5-18
- *LDFD ARM pseudo-instruction* on page 5-19
- *LDFS ARM pseudo-instruction* on page 5-20
- *LDR ARM pseudo-instruction* on page 5-21
- *NOP ARM pseudo-instruction* on page 5-23.

See *Thumb pseudo-instructions* on page 5-24 for information on pseudo-instructions that are available in Thumb state.

5.7.1 ADR ARM pseudo-instruction

The `ADR` pseudo-instruction loads a program-relative or register-relative address into a register.

Syntax

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

where:

`condition`	is an optional condition code
`register`	is the register to load.
`expression`	is a program-relative or register-relative expression that evaluates to: • a non word-aligned address within 255 bytes • a word-aligned address within 1020 bytes. The address can be either before or after the address of the instruction or the base register (see *Register-relative and program-relative expressions* on page 5-119).

Usage

`ADR` always assembles to one instruction. The assembler attempts to produce a single `ADD` or `SUB` instruction to load the address. If the address cannot be constructed in a single instruction, an error is generated and the assembly fails.

`ADR` produces position-independent code, because the address is program-relative or register-relative.

Use the `ADRL` pseudo-instruction to assemble a wider range of effective addresses.

If *`expression`* is program-relative, it must evaluate to an address in the same code section as the `ADR` pseudo-instruction. Otherwise, it may be out of range after linking.

Example

```
start   MOV     r0,#10
        ADR     r4,start    ; => SUB r4,pc,#0xc
```

5.7.2 ADRL ARM pseudo-instruction

The `ADRL` pseudo-instruction loads a program-relative or register-relative address into a register. It is similar to the `ADR` pseudo-instruction. `ADRL` can load a wider range of addresses than `ADR` because it generates two data processing instructions.

——— Note ———

`ADRL` is not available when assembling Thumb instructions. Use it only in ARM code.

Syntax

`ADRL{condition} register,expression`

where:

`condition`	is an optional condition code.
`register`	is the register to load.
`expression`	is a relative expression that evaluates to: • a non word-aligned address within ±64KB • a word-aligned address within ±256KB. The address can be relative to either the current instruction or a base register (see *Register-relative and program-relative expressions* on page 5-119).

Usage

`ADRL` always assembles to two instructions. Even if the address can be reached in a single instruction, a second, redundant instruction is produced.

If the assembler cannot construct the address in two instructions, it generates an error message and the assembly fails. See *LDR ARM pseudo-instruction* on page 5-21 for information on loading a wider range of addresses (see also the assembly language chapter in *ADS Developer Guide*).

`ADRL` produces position-independent code, because the address is program-relative or register-relative.

If *`expression`* is program-relative, it must evaluate to an address in the same code section as the `ADRL` pseudo-instruction. Otherwise, it may be out of range after linking.

-

Example

```
start   MOV     r0,#10
        ADRL    r4,start + 60000      ; => ADD r4,pc,#0xe800
                                      ;    ADD r4,r4,#0x254
```

5.7.3 FLDD ARM pseudo-instruction

The `FLDD` pseudo-instruction loads a VFP floating-point register with a double-precision floating-point constant.

Note

You can use `FLDD` only if the command line option `-fpu` is set to `vfp`.

This section describes the `FLDD` *pseudo*-instruction only. See the *ARM Architecture Reference Manual* for information on the `FLDD` *instruction*.

Syntax

`FLDD{`*`condition`*`}` *`fp-register,=fp-literal`*

where:

`condition`	is an optional condition code.
`fp-register`	is the floating-point register to be loaded.
`fp-literal`	is a double-precision floating-point literal (see *Floating-point literals* on page 5-118).

Usage

The range for double-precision numbers is:

- maximum 1.79769313486231571e+308
- minimum 2.22507385850720138e–308.

The assembler places the constant in a literal pool and generates a program-relative `FLDD` instruction to read the constant from the literal pool. Two words are used to store the constant in the literal pool.

The offset from pc to the constant must be less than 1KB. You are responsible for ensuring that there is a literal pool within range. See *LTORG directive* on page 5-91 for more information.

Example

```
        FLDD    d1,=3.12E106    ; loads 3.12E106 into d1
```

5.7.4 FLDS ARM pseudo-instruction

The `FLDS` pseudo-instruction loads a VFP floating-point register with a single-precision floating-point constant.

Note

You can use `FLDS` only if the command line option `-fpu` is set to `vfp`.

This section describes the `FLDS` *pseudo*-instruction only. See the *ARM Architecture Reference Manual* for information on the `FLDS` *instruction*.

Syntax

`FLDS{`*`condition`*`}` *`fp-register`*`,=`*`fp-literal`*

where:

`condition`	is an optional condition code.
`fp-register`	is the floating-point register to be loaded.
`fp-literal`	is a single-precision floating-point literal (see *Floating-point literals* on page 5-118).

Usage

The range for single-precision values is:

- maximum 3.40282347e+38
- minimum 1.17549435e–38.

The assembler places the constant in a literal pool and generates a program-relative `FLDS` instruction that reads the constant from the literal pool.

The offset from the pc to the constant must be less than 1KB. You are responsible for ensuring that there is a literal pool within range. See *LTORG directive* on page 5-91 for more information.

Example

```
        FLDS    s1,=3.12E-6    ; loads 3.12E-6 into s1
```

5.7.5 LDFD ARM pseudo-instruction

The `LDFD` pseudo-instruction loads an FPA floating-point register with a double-precision floating-point constant.

Note

You can use `LDFD` only if the command line option `-fpu` is set to `fpa`.

This section describes the `LDFD` *pseudo*-instruction only.

Syntax

`LDFD{`*`condition`*`}` *`fp-register`*`,=`*`fp-literal`*

where:

`condition`	is an optional condition code.
`fp-register`	is the floating-point register to be loaded.
`fp-literal`	is a double-precision floating-point literal (see *Floating-point literals* on page 5-118).

Usage

The range for double-precision numbers is:
- maximum 1.79769313486231571e+308
- minimum 2.22507385850720138e–308.

The assembler places the constant in a literal pool and generates a program-relative `LDFD` instruction to read the constant from the literal pool. Two words are used to store the constant in the literal pool.

The offset from pc to the constant must be less than 1KB. You are responsible for ensuring that there is a literal pool within range. See *LTORG directive* on page 5-91 for more information.

Example

```
        LDFD    f1,=3.12E106    ; loads 3.12E106 into f1
```

5.7.6 LDFS ARM pseudo-instruction

The `LDFS` pseudo-instruction loads an FPA floating-point register with a single-precision floating-point constant.

—— Note ——

You can use `LDFS` only if the command line option `-fpu` is set to `fpa`.

This section describes the `LDFS` *pseudo*-instruction only.

Syntax

LDFS{*condition*} *fp-register*,=*fp-literal*

where:

condition	is an optional condition code.
fp-register	is the floating-point register to be loaded.
fp-literal	is a single-precision floating-point literal (see *Floating-point literals* on page 5-118).

Usage

The range for single-precision values is:

- maximum 3.40282347e+38
- minimum 1.17549435e–38.

The assembler places the constant in a literal pool and generates a program-relative `LDFS` instruction that reads the constant from the literal pool.

The offset from the pc to the constant must be less than 1KB. You are responsible for ensuring that there is a literal pool within range. See *LTORG directive* on page 5-91 for more information.

Example

```
        LDFS    f1,=3.12E-6    ; loads 3.12E-6 into f1
```

5.7.7 LDR ARM pseudo-instruction

The `LDR` pseudo-instruction loads a register with either:

- a 32-bit constant value
- an address.

Note

This section describes the `LDR` *pseudo*-instruction only. See the *ARM Architecture Reference Manual* for information on the `LDR` *instruction*.

Syntax

```
LDR{condition} register,=[expression | label-expression]
```

where:

`condition` is an optional condition code.

`register` is the register to be loaded.

`expression`

evaluates to a numeric constant:

- the assembler generates a `MOV` or `MVN` instruction, if the value of *`expression`* is within range
- if the value of *`expression`* is *not* within range of a `MOV` or `MVN` instruction, the assembler places the constant in a literal pool and generates a program-relative `LDR` instruction that reads the constant from the literal pool.

`label-expression`

is a program-relative or external expression. The assembler places the value of *`label-expression`* in a literal pool and generates a program-relative `LDR` instruction that loads the value from the literal pool.

If *`label-expression`* is an external expression, or is not contained in the current section, the assembler places a linker relocation directive in the object file. The linker generates the address at link time.

Usage

The LDR pseudo-instruction is used for two main purposes:

- To generate literal constants when an immediate value cannot be moved into a register because it is out of range of the MOV and MVN instructions
- To load a program-relative or external address into a register. The address remains valid regardless of where the linker places the ELF section containing the LDR.

  Note

  An address loaded in this way is fixed at link time, so the code is *not* position-independent.

The offset from the pc to the value in the literal pool must be less than 4KB. You are responsible for ensuring that there is a literal pool within range. See *LTORG directive* on page 5-91 for more information.

See the assembly language chapter in *ADS Developer Guide* for a more detailed explanation of how to use LDR, and for more information on MOV and MVN.

Example

```
        LDR     r3,=0xff0     ; loads 0xff0 into r3
                              ; =>  MOV r3,#0xff0

        LDR     r1,=0xfff     ; loads 0xfff into r1
                              ; =>  LDR r1,[pc,offset_to_litpool]
                              ;     ...
                              ;     litpool DCD 0xfff

        LDR     r2,=place     ; loads the address of
                              ; place into r2
                              ; =>  LDR r2,[pc,offset_to_litpool]
                              ;     ...
                              ;     litpool DCD place
```

5.7.8 NOP ARM pseudo-instruction

`NOP` generates the preferred ARM no-operation code. This is:

```
MOV r0,r0
```

Syntax

```
NOP
```

Usage

`NOP` cannot be used conditionally. Not executing a no-operation is the same as executing it, so conditional execution is not required.

ALU status flags are unaltered by `NOP`.

5.8 Thumb pseudo-instructions

The ARM assembler supports a number of pseudo-instructions that are translated into the appropriate combination of ARM or Thumb instructions at assembly time.

The pseudo-instructions that are available in Thumb state are in the following sections:

- *ADR Thumb pseudo-instruction* on page 5-25
- *LDR Thumb pseudo-instruction* on page 5-26
- *MOV Thumb pseudo-instruction* on page 5-28
- *NOP Thumb pseudo-instruction* on page 5-29.

See *ARM pseudo-instructions* on page 5-13 for information on pseudo-instructions that are available in ARM state.

5.8.1 ADR Thumb pseudo-instruction

The `ADR` pseudo-instruction loads a program-relative or register-relative address into a register.

Syntax

`ADR` *`register`*`,` *`expression`*

where:

`register` is the register to load.

`expression`

is a program-relative or register-relative expression. The offset must be positive and less than 1KB. *`expression`* must be defined locally, it cannot be imported.

See *MAP or ^ directive* on page 5-95 for more information on register-relative expressions.

Usage

In Thumb state, `ADR` can generate word-aligned addresses only. Use the `ALIGN` directive to ensure that *`expression`* is aligned.

If *`expression`* is program-relative, it must evaluate to an address in the same code section as the `ADR` pseudo-instruction. There is no guarantee that the address will be within range after linking if it resides in another ELF section.

Example

```
        ADR     r4,txampl    ; => ADD r4,pc,#nn
        ; code
        ALIGN
txampl  DCW     0,0,0,0
```

5.8.2 LDR Thumb pseudo-instruction

The `LDR` pseudo-instruction loads a low register with either:

- a 32-bit constant value
- an address.

Note

This section describes the `LDR` *pseudo*-instruction only. See the *ARM Architecture Reference Manual* for information on the `LDR` *instruction*.

Syntax

```
LDR register, =[expression | label-expression]
```

where:

register is the register to be loaded. `LDR` can access the low registers (r0-r7) only.

expression

evaluates to a numeric constant:

- if the value of *expression* is within range of a `MOV` instruction, the assembler generates the instruction
- if the value of *expression* is *not* within range of a `MOV` instruction, the assembler places the constant in a literal pool and generates a program-relative `LDR` instruction that reads the constant from the literal pool.

label-expression

is a program-relative or external expression. The assembler places the value of *label-expression* in a literal pool and generates a program-relative `LDR` instruction that loads the value from the literal pool.

If *label-expression* is an external expression, or is not contained in the current section, the assembler places a linker relocation directive in the object file. The linker ensures that the correct address is generated at link time.

The offset from the pc to the value in the literal pool must be positive and less than 1KB. You are responsible for ensuring that there is a literal pool within range. See *LTORG directive* on page 5-91 for more information.

Usage

The `LDR` pseudo-instruction is used for two main purposes:

- To generate literal constants when an immediate value cannot be moved into a register because it is out of range of the `MOV` instruction.
- To load a program-relative or external address into a register. The address remains valid regardless of where the linker places the ELF section containing the `LDR`.

See the assembly language chapter in *ADS Developer Guide* for a more detailed explanation of how to use `LDR`, and for more information on `MOV`.

Example

```
        LDR     r1, =0xfff      ; loads 0xfff into r1
                                ;
        LDR     r2, =labelname  ; loads the address of
                                ; labelname into r2
```

5.8.3 MOV Thumb pseudo-instruction

The Thumb `MOV` *pseudo*-instruction moves the value of a low register to another low register (r0-r7).

The Thumb `MOV` *instruction* cannot move values from one low register to another.

Note

The `ADD` immediate instruction generated by the assembler has the side-effect of updating the condition codes.

Syntax

`MOV` *`Rd`*`,`*`Rs`*

where:

`Rd`	is the destination register.
`Rs`	is the source register.

Usage

The `MOV` pseudo-instruction uses an `ADD` immediate instruction with a zero immediate value.

See the *ARM Architecture Reference Manual* for more information on the Thumb `MOV` instruction.

Example

```
    MOV     Rd, Rs  ; generates the opcode for ADD Rd, Rs, #0
```

5.8.4 NOP Thumb pseudo-instruction

`NOP` generates the preferred Thumb no-operation instruction. This is:

```
MOV r8,r8
```

Syntax

The syntax for `NOP` is:

```
NOP
```

Usage

ALU status flags are unaltered by `NOP`.

5.9 Symbols

You can use symbols to represent variables, addresses, and numeric constants. Symbols representing addresses are also called labels. See:

- *Variables* on page 5-31
- *Numeric constants* on page 5-31
- *Labels* on page 5-33.

5.9.1 Symbol naming rules

The following general rules apply to symbol names:

- You can use uppercase letters, lowercase letters, numeric characters, or the underscore character in symbol names.
- Do not use numeric characters for the first character of symbol names, except in local labels (see *Local labels* on page 5-34).
- Symbol names are case-sensitive.
- All characters in the symbol name are significant.
- Symbol names must be unique within their scope.
- Symbols must not use built-in variable names or predefined symbol names (see *Predefined register and coprocessor names* on page 5-10 and *Built-in variables* on page 5-12).
- Symbols should not use the same name as instruction mnemonics or directives. If you need to use the same name as an instruction mnemonic or directive, use double bars to delimit the symbol name. For example:

  ```
  ||ASSERT||
  ```

  The bars are not part of the symbol. You cannot use bars, semicolons, or newlines within the bars.

If you need to use a wider range of characters in symbols, use single bars to delimit the symbol name.

The bars are not part of the symbol. You cannot use bars, semicolons, or newlines within the bars.

5.9.2 Variables

The value of a variable can be changed as assembly proceeds. Variables are of three types:

- numeric
- logical
- string.

The type of a variable cannot be changed.

The range of possible values of a numeric variable is the same as the range of possible values of a numeric constant or numeric expression (see *Numeric constants* on page 5-31 and *Numeric expressions* on page 5-116).

The possible values of a logical variable are `{TRUE}` or `{FALSE}` (see *Logical expressions* on page 5-119).

The range of possible values of a string variable is the same as the range of values of a string expression (see *String expressions* on page 5-115).

Use the `GBLA`, `GBLL`, `GBLS`, `LCLA`, `LCLL`, and `LCLS` directives to declare symbols representing variables, and assign values to them using the `SETA`, `SETL`, and `SETS` directives. See:

- *GBLA directive* on page 5-79
- *GBLL directive* on page 5-80
- *GBLS directive* on page 5-81
- *LCLA directive* on page 5-88
- *LCLL directive* on page 5-89
- *LCLS directive* on page 5-90
- *SETA directive* on page 5-103
- *SETL directive* on page 5-104
- *SETS directive* on page 5-105.

5.9.3 Numeric constants

Numeric constants are 32-bit integers. You can set them using unsigned numbers in the range 0 to $2^{32} - 1$, or signed numbers in the range -2^{31} to $2^{31} - 1$. However, the assembler makes no distinction between $-n$ and $2^{32} - n$. Relational operators such as >= use the unsigned interpretation. This means that 0 > –1 is `{FALSE}`.

Use the `EQU` directive to define constants (see *EQU or * directive* on page 5-64). You cannot change the value of a numeric constant after you define it.

See also *Numeric expressions* on page 5-116 and *Numeric literals* on page 5-117.

5.9.4 Assembly time substitution of variables

You can use a string variable for a whole line of assembly language, or any part of a line. Use the variable with a `$` prefix in the places where the value is to be substituted for the variable. The dollar character instructs the assembler to substitute the string into the source code line before checking the syntax of the line.

Numeric and logical variables can also be substituted. The current value of the variable is converted to a hexadecimal string (or `T` or `F` for logical variables) before substitution.

Use a dot to mark the end of the variable name if the following character would be permissible in a symbol name (see *Symbol naming rules* on page 5-30). You must set the contents of the variable before you can use it.

If you need a `$` that you do not want to be substituted, use `$$`. This is converted to a single `$`.

You can include a variable with a `$` prefix in a string. Substitution occurs in the same way as anywhere else.

Substitution does not occur within vertical bars, except that vertical bars within double quotes do not affect substitution.

Examples

```
    ; straightforward substitution
        GBLS    add4ff
        ;
add4ff  SETS    "ADD  r4,r4,#0xFF"    ; set up add4ff
        $add4ff.00                    ; invoke add4ff
        ; this produces
        ADD  r4,r4,#0xFF00

    ; elaborate substitution
            GBLS    s1
            GBLS    s2
            GBLS    fixup
            GBLA    count
            ;
count       SETA    14
s1          SETS    "a$$b$count" ; s1 now has value a$b0000000E
s2          SETS    "abc"
fixup       SETS    "|xy$s2.z|"  ; fixup now has value |xyabcz|
|C$$code|    MOV     r4,#16      ; but the label here is C$$code
```

-

5.9.5 Labels

Labels are symbols representing the addresses in memory of instructions or data. They may be program-relative, register-relative, or absolute.

Program-relative labels

These represent the program counter plus or minus a numeric constant. Use them as targets for branch instructions, or to access small items of data embedded in code sections. You can define program-relative labels using a label on an instruction or on one of the Define Constant directives. See:

- *DCB or = directive* on page 5-47
- *DCD or & directive* on page 5-48
- *DCDU directive* on page 5-50
- *DCFD directive* on page 5-51
- *DCFDU directive* on page 5-52
- *DCFS directive* on page 5-53
- *DCFSU directive* on page 5-54
- *DCW directive* on page 5-58
- *DCWU directive* on page 5-59.

Register-relative labels

These represent a named register plus a numeric constant. They are most often used to access data in data sections. You can define them with a storage map. You can use the `EQU` directive to define additional register-relative labels, based on labels defined in storage maps. See:

- *DCDO directive* on page 5-49
- *MAP or ^ directive* on page 5-95
- *SPACE or % directive* on page 5-107
- *EQU or * directive* on page 5-64.

Absolute addresses

These are numeric constants. They are integers in the range 0 to 2^{32}–1. They See memory directly.

-

5.9.6 Local labels

A local label is a number in the range 0-99, optionally followed by a name. The same number can be used for more than one local label in an ELF section.

Local labels are used for instructions that are the target for branches. You cannot use them for data. Typically they are used for loops and conditional code within a routine, or for small subroutines that are only used locally. They are particularly useful in macros (see *MACRO directive* on page 5-92).

Use the `ROUT` directive to limit the scope of local labels (see *ROUT directive* on page 5-102). A reference to a local label refers to a matching label within the same scope. If there is no matching label within the scope in either direction, the assembler generates an error message and the assembly fails.

You can use the same number for more than one local label even within the same scope. By default, the assembler links a local label reference to:

- the most recent local label of the same number, if there is one within the scope
- the next following local label of the same number, if there is not a preceding one within the scope.

Use the optional parameters to modify this search pattern if required.

Syntax

The syntax of a local label is:

`n{routname}`

The syntax of a reference to a local label is:

`%{F|B}{A|T}n{routname}`

where:

n	is the number of the local label.
routname	is the name of the current scope.
`%`	introduces the reference.
`F`	instructs the assembler to search forwards only.
`B`	instructs the assembler to search backwards only.
`A`	instructs the assembler to search all macro levels.
`T`	instructs the assembler to look at this macro level only.

If neither `F` or `B` is specified, the assembler searches backwards first, then forwards.

If neither `A` or `T` is specified, the assembler searches all macros from the current level to the top level, but does not search lower level macros.

If *routname* is specified in either a label or a reference to a label, the assembler checks it against the name of the nearest preceding `ROUT` directive. If it does not match, the assembler generates an error message and the assembly fails.

5.10 Directives

The assembler provides directives to support:

- data structure definitions and allocation of space for data
- partitioning of files into logical subdivisions
- error reporting and control of assembly listing
- definition of symbols
- conditional and repetitive assembly, and inclusion of subsidiary files.

See Table 5-2 on page 5-2 to locate individual directives within this section. The directives are described in the following sections in alphabetical order.

5.10.1 Nesting directives

`MACRO` definitions, `WHILE...WEND` loops, `IF...ENDIF` conditions and `GET` or `INCLUDE` directives can be nested within themselves or within each other to a total depth of 256.

5.10.2 ALIGN directive

The `ALIGN` directive aligns the current location to a specified boundary by padding with zeroes.

Syntax

ALIGN {*expression*{,*offset*}}

where:

expression	can be any power of 2 from 2^0 to 2^{31}.
offset	can be any numeric expression.

The current location is aligned to the next address of the form:

*offset + n * expression*

If *expression* is not specified, `ALIGN` sets the current location to the next word boundary.

Usage

Use `ALIGN` to ensure that your code and data is correctly aligned. As a general rule it is safer to use `ALIGN` frequently through your code.

Use `ALIGN` to ensure that Thumb addresses are word-aligned when required. For example, the `ADR` Thumb pseudo-instruction can only load addresses that are word aligned.

Use `ALIGN` when data definition directives appear in code sections. When data definition directives (`DCB`, `DCW`, `DCWU`, `DCDU` and `SPACE`) are used in code sections, the program counter does not necessarily point to a word boundary. When the assembler encounters the next instruction mnemonic it inserts up to 3 bytes, if required, to ensure that the instruction is:

- word-aligned in ARM state
- halfword-aligned in Thumb state.

In this case, a label that appears on a source line by itself does not address the following instruction. Use an `ALIGN` directive before the label to ensure that the label addresses the following instruction. You can use `ALIGN 2` to align on a halfword (2-byte) boundary in Thumb code.

Use `ALIGN` with a coarser setting to take advantage of caches on some ARM processors. For example, the ARM940T has a cache with 16-byte lines. Use `ALIGN 16` to align function entries on 16-byte boundaries and maximize the efficiency of the cache.

Alignment is relative to the start of the ELF section where the routine is located. You must ensure that the section is also aligned to the same, or coarser, boundaries. The `ALIGN` attribute on the `AREA` directive is specified differently (see *AREA directive* on page 5-39 and the example below).

Examples

```
        AREA    Example, CODE, READONLY
start   LDR     r6,=label1
        ; code
        MOV     pc,lr
label1  DCB     1            ; pc now misaligned
        ALIGN                ; ensures that label1 addresses
                             ; the following instruction.
subr1   MOV r5,#0x5


        AREA    cacheable, CODE, ALIGN=3
rout1   ; code               ; aligned on 8-byte boundary
        ; code
        MOV     pc,lr        ; aligned only on 4-byte boundary
        ALIGN   8            ; now aligned on 8-byte boundary
rout2   ; code


        AREA    OffsetExample, CODE
        DCB     1            ; This example places the two
        ALIGN   4,3          ; bytes in the first and fourth
        DCB     1            ; bytes of the same word.
```

5.10.3 AREA directive

The `AREA` directive instructs the assembler to assemble a new code or data section. Sections are independent, named, indivisible chunks of code or data that are manipulated by the linker. See the assembly language chapter in *ADS Developer Guide* for more information.

Syntax

`AREA` *`sectionname`*`{,`*`attr`*`}{,`*`attr`*`}...`

where:

`sectionname`

is the name that the section is to be given.

You can choose any name for your sections. However, names starting with a digit must be enclosed in bars or a missing section name error is generated. For example, `|1_DataArea|`.

Certain names are conventional. For example, `|.text|` is used for code sections produced by the C compiler, or for code sections otherwise associated with the C library.

`attr` are one or more comma-delimited section attributes. Valid attributes are:

`ALIGN=`*`expression`*

By default, ELF sections are aligned on a 4-byte boundary.

`expression` can have any integer value between 2 and 31. The section is aligned on a $2^{expression}$-byte boundary. For example, if *expression* is 10, the section is aligned on a 1KB boundary. *This is not the same as the way that the* `ALIGN` *directive is specified.* See *ALIGN directive* on page 5-37.

`ASSOC=`*`section`*

`section` specifies an associated ELF section. *`sectionname`* must be included in any link that includes *`section`*.

`CODE` Contains machine instructions. `READONLY` is the default.

`COMDEF` Is a common section definition. This ELF section may contain code or data. It must be identical to any other section of the same name in other source files.

Identical ELF sections with the same name are overlaid in the same section of memory by the linker. If any are different, the linker generates a warning and does not overlay the sections.

`COMMON`	Is a common data section. You must not define any code or data in it. It is initialized to zeroes by the linker. All common sections with the same name are overlaid in the same section of memory by the linker. They do not all need to be the same size. The linker allocates as much space as is required by the largest common section of each name.
`DATA`	Contains data, not instructions. `READWRITE` is the default.
`NOINIT`	Indicates that the data section is uninitialized, or initialized to zero. It contains only space reservation directives (`DCB`, `DCD`, `DCDU`, `DCQ`, `DCQU`, `DCW`, `DCWU`, or `SPACE`), with no initialized values.
`READONLY`	Indicates that this section should not be written to.
`READWRITE`	Indicates that this section may be read from and written to.

Usage

Use the `AREA` directive to subdivide your source file into ELF sections. You can use the same name in more than one `AREA` directive. All areas with the same name are placed in the same ELF section.

You should normally use separate ELF sections for code and data. Large programs can usually be conveniently divided into several code sections. Large independent data sets are also usually best placed in separate sections.

The scope of local labels is defined by `AREA` directives, optionally subdivided by `ROUT` directives (see *Local labels* on page 5-34 and *ROUT directive* on page 5-102).

There must be at least one `AREA` directive for an assembly. If no `AREA` directive is specified, the assembler generates an ELF section with the name `|$$$$$$$|`, and produces a diagnostic message. This limits the number of error messages caused by the missing directive, but does not lead to a successful assembly.

Example

The following example defines a read-only code section named Example.

```
    AREA    Example,CODE,READONLY   ; An example code section.
            ; code
```

5.10.4 ASSERT directive

The `ASSERT` directive generates an error message during the second pass of the assembly if a given assertion is false.

Syntax

`ASSERT` *`logical-expression`*

where:

`logical-expression`

is an assertion that can evaluate to either `{TRUE}` or `{FALSE}`.

Usage

Use `ASSERT` to ensure that any necessary condition is met during assembly.

If the assertion is false an error message is generated and assembly fails.

See also *INFO or ! directive* on page 5-86.

Example

```
        ASSERT  label1 <= label2    ; Tests if the address
                                    ; represented by label1
                                    ; is <= the address
                                    ; represented by label2.
```

-

5.10.5 CN directive

The CN directive defines a name for a coprocessor register.

Syntax

name CN *numeric-expression*

where:

name is the name to be defined for the coprocessor register.

numeric-expression
evaluates to a coprocessor register number from 0 to 15.

Usage

Use CN to allocate convenient names to registers, to help you remember what you use each register for. Be careful to avoid conflicting uses of the same register under different names.

The names c0 to c15 are predefined.

Example

```
power    CN  6        ; defines power as a symbol for
                      ; coprocessor register 6
```

5.10.6 CODE16 directive

The `CODE16` directive instructs the assembler to interpret subsequent instructions as 16-bit Thumb instructions.

Syntax

```
CODE16
```

Usage

In files that contain a mixture of ARM and Thumb code, use `CODE16` when changing from ARM state to Thumb state. `CODE16` must precede any Thumb code.

The assembler inserts a byte of padding, if necessary, to bring following Thumb code into halfword alignment. `CODE16` does not assemble to an instruction that changes the state. It only instructs the assembler to assemble Thumb instructions.

See also *CODE32 directive* on page 5-44.

Example

This example shows how `CODE16` can be used to branch from ARM to Thumb instructions.

```
        AREA    ThumbEx, CODE, READONLY

                                ; This section starts in ARM state
        LDR     r0,=start+1     ; Load the address and set the
                                ; least significant bit
        BX      r0              ; Branch and exchange
                                ; instruction sets

                                ; Not necessarily in same section

        CODE16                  ; Following instructions are Thumb
start   MOV     r1,#10          ; Thumb instructions
```

-

5.10.7 CODE32 directive

The `CODE32` directive instructs the assembler to interpret subsequent instructions as 32-bit ARM instructions.

Syntax

```
CODE32
```

Usage

In files that contain a mixture of ARM and Thumb code, use `CODE32` when branching from Thumb state to ARM state. `CODE32` precedes the ARM code.

The assembler inserts up to three bytes of padding, if necessary, to bring following ARM code into word alignment. `CODE32` does not assemble to an instruction that changes the state. It only instructs the assembler to assemble ARM instructions.

See also *CODE16 directive* on page 5-43.

Example

```
        CODE16             ; Start this section in Thumb state

        AREA    ThumbEx, CODE, READONLY

        MOV     r1,#10     ; Thumb instructions
        LDR     r0,=goarm  ; Load the address and leave the
                           ; least significant bit clear.
        BX      r0         ; Branch and exchange instruction
                           ; sets

                           ; Not necessarily in the same section
        CODE32             ; Following instructions are ARM
goarm   MOV     r4,#15     ; ARM instructions
```

5.10.8 CP directive

The `CP` directive defines a name for a specified coprocessor. The coprocessor number must be within the range 0 to 15.

Syntax

`name` `CP` *`numeric-expression`*

where:

`name` is the name to be assigned to the coprocessor. *`name`* cannot be the same as any of the predefined names listed in *Predefined register and coprocessor names* on page 5-10.

`numeric-expression`
evaluates to a coprocessor number from 0 to 15.

Usage

Use `CP` to allocate convenient names to coprocessors, to help you to remember what you use each one for. Be careful to avoid conflicting uses of the same coprocessor under different names.

The names p0 to p15 are predefined for coprocessors 0 to 15.

Example

```
dmu    CP   6       ; defines dmu as a symbol for
                    ; coprocessor 6
```

5.10.9 DATA directive

The DATA directive informs the assembler that a label is a *data-in-code* label. This means that the label is the address of data within a code segment.

Syntax

label DATA

where:

label	is the label of the data definition. The DATA directive must be on the same line as *label*.

Usage

You *must* use the DATA directive when you define data in a Thumb code section with any of the data-defining directives such as DCD, DCB, and DCW.

When the linker relocates a label in a Thumb code section, it assumes that the label represents the address of a Thumb routine. The linker adds one to the value of the label so that the processor is switched to Thumb state if the routine is called with a BX instruction.

If a label represents the address of data in a Thumb code section, you do not want the linker to add one to the label. The DATA directive marks the label as pointing to data within a code section and the linker does not add one to the value.

You can use DATA to mark data-in-code in ARM code areas. The DATA directive is ignored by the assembler in ARM code areas.

Example

```
            AREA    example, CODE
Thumb_fn    ; code
            ; code
            MOV     pc, lr

Thumb_Data  DATA
            DCB     1, 3, 4
```

-

5.10.10 DCB or = directive

The `DCB` directive allocates one or more bytes of memory, and defines the initial runtime contents of the memory. `=` is a synonym for `DCB`.

Syntax

```
{label} DCB expression{,expression}...
```

where:

expression

is either:

- A numeric expression that evaluates to an integer in the range –128 to 255 (see *Numeric expressions* on page 5-116).
- A quoted string. The characters of the string are loaded into consecutive bytes of store.

Usage

You *must* use the `DATA` directive if you use `DCB` to define labeled data within Thumb code. See *DATA directive* on page 5-46 for more information.

If `DCB` is followed by an instruction, use an `ALIGN` directive to ensure that the instruction is aligned. See *ALIGN directive* on page 5-37 for more information.

See also:

- *DCD or & directive* on page 5-48
- *DCDU directive* on page 5-50
- *DCQ directive* on page 5-56
- *DCQU directive* on page 5-57
- *DCW directive* on page 5-58
- *DCWU directive* on page 5-59
- *SPACE or % directive* on page 5-107.

Example

Unlike C strings, ARM assembler strings are not null-terminated. You can construct a null-terminated C string using `DCB` as follows:

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

5.10.11 DCD or & directive

The `DCD` directive allocates one or more words of memory, aligned on 4-byte boundaries, and defines the initial runtime contents of the memory. `&` is a synonym for `DCD`.

Syntax

```
{label} DCD expression{,expression}
```

where:

expression

is either:

- a numeric expression (see *Numeric expressions* on page 5-116).
- a program-relative expression.

Usage

`DCD` inserts up to 3 bytes of padding before the first defined word, if necessary, to achieve 4-byte alignment. Use `DCDU` if you do not require alignment.

See also:

- *DCB or = directive* on page 5-47
- *DCW directive* on page 5-58
- *DCQ directive* on page 5-56
- *DCDU directive* on page 5-50
- *SPACE or % directive* on page 5-107.

Example

```
data1   DCD     1,5,20          ; Defines 3 words containing
                                ; decimal values 1, 5, and 20

data2   DCD     mem06           ; Defines 1 word containing the
                                ; address of the label mem06

data3   DCD     glb + 4         ; Defines 1 word containing
                                ; 4 + the value of glb
```

5.10.12 DCDO directive

The `DCDO` directive allocates one or more words of memory, aligned on 4-byte boundaries, and defines the initial runtime contents of the memory as an offset from the *static base register*, sb (r9).

Syntax

`{`*`label`*`} DCDO `*`expression`*`{,`*`expression`*`}...`

where:

`expression`

is a register-relative expression or label. The base register must be sb.

Usage

Use `DCDO` to allocate space in memory for static base register relative relocatable addresses.

See also:

- *DCD or & directive* on page 5-48
- *Labels* on page 5-33
- *MAP or ^ directive* on page 5-95.

Example

```
        IMPORT  externsym
        DCDO    externsym   ; 32-bit word relocated by offset of
                            ; externsym from base of SB section.
```

5.10.13 DCDU directive

The DCDU directive allocates one or more words of memory, not necessarily aligned, and defines the initial runtime contents of the memory.

Syntax

{*label*} DCDU *expression*{,*expression*}...

where:

expression

is either:

- A numeric expression (see *Numeric expressions* on page 5-116).
- A program-relative expression.

Usage

Use DCDU to define data words with arbitrary alignment.

If DCDU is followed by code, use an ALIGN directive to ensure that the instructions are word aligned. See *ALIGN directive* on page 5-37 for more information.

DCDU does not insert padding when preceding code is unaligned. Use DCD if you require alignment.

See also:

- *DCB or = directive* on page 5-47
- *DCD or & directive* on page 5-48
- *DCQU directive* on page 5-57
- *DCWU directive* on page 5-59
- *SPACE or % directive* on page 5-107.

Example

```
        AREA    MyData, DATA, READWRITE
        DCB     255           ; Now misaligned ...
data1   DCDU    1,5,20        ; Defines 3 words containing
                              ; 1, 5 and 20, not word aligned
```

-

5.10.14 DCFD directive

The `DCFD` directive allocates memory for word-aligned double-precision floating-point numbers, and defines the initial runtime contents of the memory. Double-precision numbers occupy two words and must be word aligned to be used in arithmetic operations.

Syntax

```
{label} DCFD fp-literal{,fp-literal}...
```

where:

`fp-literal`

is a double-precision floating-point literal (see *Floating-point literals* on page 5-118).

Usage

The assembler inserts up to three bytes of padding before the first defined number, if necessary, to achieve 4-byte alignment. Use `DCFDU` if you do not require alignment.

The word order used when converting *`fp-literal`* to internal form is controlled by the floating-point architecture selected.

The range for double-precision numbers is:

- maximum 1.79769313486231571e+308
- minimum 2.22507385850720138e–308.

See also:

- *DCFDU directive* on page 5-52
- *DCFS directive* on page 5-53
- *DCFSU directive* on page 5-54.

Examples

```
        DCFD    1E308,-4E-100
        DCFD    10000,-.1,3.1E26
```

5.10.15 DCFDU directive

The `DCFDU` directive allocates eight bytes of memory for arbitrarily aligned double-precision floating-point numbers, and defines the initial runtime contents of the memory.

Syntax

```
{label} DCFDU fp-literal{,fp-literal}...
```

where:

`fp-literal`

is a double-precision floating-point literal (see *Floating-point literals* on page 5-118).

Usage

`DCFDU` defines floating-point values with arbitrary alignment. Use `DCFD` if you require alignment.

The word order used when converting *`fp-literal`* to internal form is controlled by the floating-point architecture selected.

The range for double-precision numbers is:
- maximum 1.79769313486231571e+308
- minimum 2.22507385850720138e–308.

See also:
- *DCFD directive* on page 5-51
- *DCFS directive* on page 5-53
- *DCFSU directive* on page 5-54.

Examples

```
        DCFDU   1E308,-4E-100
        DCFDU   100,-.1,3.1E26
```

5.10.16 DCFS directive

The `DCFS` directive allocates memory for word-aligned single-precision floating-point numbers, and defines the initial runtime contents of the memory. Single-precision numbers occupy one word and must be word aligned to be used in arithmetic operations.

Syntax

```
{label} DCFS fp-literal{,fp-literal}...
```

where:

`fp-literal`
: is a single-precision floating-point literal (see *Floating-point literals* on page 5-118).

Usage

`DCFS` inserts up to three bytes of padding before the first defined number, if necessary to achieve 4-byte alignment. Use `DCFSU` if you do not require alignment.

The range for single-precision values is:

- maximum 3.40282347e+38
- minimum 1.17549435e–38.

See also:

- *DCFD directive* on page 5-51
- *DCFDU directive* on page 5-52
- *DCFSU directive* on page 5-54.

Example

```
        DCFS    1E3,-4E-9
        DCFS    1.0,-.1,3.1E6
```

5.10.17 DCFSU directive

The DCFSU directive allocates memory for arbitrarily aligned single-precision floating-point numbers, and defines the initial runtime contents of the memory.

Syntax

```
{label} DCFSU fp-literal{,fp-literal}...
```

where:

fp-literal

is a single-precision floating-point literal (see *Floating-point literals* on page 5-118).

Usage

Use DCFSU to define floating-point values with arbitrary alignment.

DCFSU does not insert padding when preceding data is unaligned. Use DCFS if you require alignment.

The range for single-precision values is:

- maximum 3.40282347e+38
- minimum 1.17549435e–38.

See also:

- *DCFD directive* on page 5-51
- *DCFDU directive* on page 5-52
- *DCFS directive* on page 5-53.

Example

```
        DCFSU   1E3,-4E-9
        DCFSU   1.0,-.1,3.1E6
```

5.10.18 DCI

In ARM code, the `DCI` directive allocates one or more words of memory, aligned on 4-byte boundaries, and defines the initial runtime contents of the memory.

In Thumb code, the `DCI` directive allocates one or more halfwords of memory, aligned on 2-byte boundaries, and defines the initial runtime contents of the memory.

Syntax

```
{label} DCI expression{,expression}
```

where:

expression	is a numeric expression (see *Numeric expressions* on page 5-116).

Usage

The `DCI` directive is very like the `DCD` or `DCW` directives, but the location is marked as code instead of data. Use `DCI` when writing macros for new instructions not supported by the version of the assembler you are using.

In ARM code, `DCI` inserts up to three bytes of padding before the first defined word, if necessary, to achieve 4-byte alignment. In Thumb code, `DCI` inserts an initial byte of padding, if necessary, to achieve 2-byte alignment.

See also *DCD or & directive* on page 5-48 and *DCW directive* on page 5-58.

Example

```
    MACRO               ; this macro translates newinstr Rd,Rm
                        ; to the appropriate machine code
    newinst     $Rd,$Rm
    DCI         0xe16f0f10 :OR: ($Rd:SHL:12) :OR: $Rm
    MEND
```

5.10.19 DCQ directive

The DCQ directive allocates one or more 8-byte blocks of memory, aligned on 4-byte boundaries, and defines the initial runtime contents of the memory.

Syntax

```
{label} DCQ {-}literal{,{-}literal}...
```

where:

literal is a 64-bit numeric literal (see *Numeric literals* on page 5-117).

The range of numbers allowed is 0 to $2^{64} - 1$.

In addition to the characters normally allowed in a numeric literal, you may prefix *literal* with a minus sign. In this case, the range of numbers allowed is -2^{63} to -1.

The result of specifying -*n* is the same as the result of specifying $2^{64} - n$..

Usage

DCQ inserts up to 3 bytes of padding before the first defined 8-byte block, if necessary, to achieve 4-byte alignment. Use DCQU if you do not require alignment.

See also:

- *DCB or = directive* on page 5-47
- *DCD or & directive* on page 5-48
- *DCQU directive* on page 5-57
- *DCW directive* on page 5-58
- *SPACE or % directive* on page 5-107.

Example

```
        AREA    MiscData, DATA, READWRITE
data    DCQ     -225,2_101     ; 2_101 means binary 101.
        DCQ     number+4       ; number must already be defined.
```

5.10.20 DCQU directive

The DCQU directive allocates one or more 8-byte blocks of memory, arbitrarily aligned, and defines the initial runtime contents of the memory.

Syntax

{*label*} DCQU {-}*literal*{,{-}*literal*}...

where:

literal	is a 64-bit numeric literal (see *Numeric literals* on page 5-117). The range of numbers allowed is 0 to $2^{64} - 1$. In addition to the characters normally allowed in a numeric literal, you may prefix *literal* with a minus sign. In this case, the range of numbers allowed is -2^{63} to -1. The result of specifying $-n$ is the same as the result of specifying $2^{64} - n$..

Usage

If an instruction follows DCQU, use an ALIGN directive to ensure that the instruction is word aligned. See *ALIGN directive* on page 5-37 for more information.

DCQU does not insert padding if preceding code is unaligned. Use DCQ if you require alignment.

See also:

- *DCB or = directive* on page 5-47
- *DCDU directive* on page 5-50
- *DCQ directive* on page 5-56
- *DCWU directive* on page 5-59
- *SPACE or % directive* on page 5-107.

Example

```
        AREA    MiscData, DATA, READWRITE
data    DCQ     -225,2_101     ; 2_101 means binary 101.
        DCQ     number+4       ; number must already be defined.
```

5.10.21 DCW directive

The DCW directive allocates one or more halfwords of memory, aligned on 2-byte boundaries, and defines the initial runtime contents of the memory.

Syntax

{*label*} DCW *expression*{,*expression*}...

where:

expression
: is a numeric expression that evaluates to an integer in the range –32768 to 65535 (see *Numeric expressions* on page 5-116).

Usage

If DCW is followed by an instruction, use an ALIGN directive to ensure that the instruction is word aligned. See *ALIGN directive* on page 5-37 for more information.

DCW inserts a byte of padding before the first defined halfword if necessary to achieve 2-byte alignment. Use DCWU if you do not require alignment.

See also:

- *DCB or = directive* on page 5-47
- *DCD or & directive* on page 5-48
- *DCQ directive* on page 5-56
- *DCWU directive* on page 5-59
- *SPACE or % directive* on page 5-107.

Example

```
        AREA    MiscData, DATA, READWRITE
data    DCW     -225,2*number          ; number must already be
        DCW     number+4               ; defined
```

5.10.22 DCWU directive

The `DCWU` directive allocates one or more unaligned halfwords of memory, and defines the initial runtime contents of the memory.

Syntax

```
{label} DCWU expression{,expression}...
```

where:

expression

is a numeric expression that evaluates to an integer in the range –32768 to 65535 (see *Numeric expressions* on page 5-116).

Usage

Use `DCWU` to define data halfwords with arbitrary alignment, in packed structures for example.

If `DCWU` is followed by code, use an `ALIGN` directive to ensure that instructions are word-aligned. See *ALIGN directive* on page 5-37 for more information.

`DCWU` does not insert padding when preceding code is unaligned. Use `DCW` if you require alignment.

See also:

- *DCB or = directive* on page 5-47
- *DCDU directive* on page 5-50
- *DCQU directive* on page 5-57
- *DCW directive* on page 5-58
- *SPACE or % directive* on page 5-107.

Example

```
        AREA    DataB2, DATA, READWRITE
oddbits DCB     1,2,3           ; now not word aligned
        DCWU    number,-255,4   ; these will each occupy two
                                ; bytes, but not necessarily
                                ; aligned
```

5.10.23 DN directive

The `DN` directive defines a name for a specified double-precision VFP register. The names d0-d15 and D0-D15 are predefined.

Syntax

`name` `DN` *`numeric-expression`*

where:

`name` is the name to be assigned to the VFP register. *`name`* cannot be the same as any of the predefined names listed in *Predefined register and coprocessor names* on page 5-10.

`numeric-expression`
evaluates to a double-precision VFP register number from 0 to 15.

Usage

Use `DN` to allocate convenient names to double-precision VFP registers, to help you to remember what you use each one for. Be careful to avoid conflicting uses of the same register under different names.

You cannot specify a vector length in a `DN` directive (see *VFP directives and notation* on page 5-11).

See also *SN directive* on page 5-106.

Example

```
energy  DN  6               ; defines energy as a symbol for
                            ; VFP double-precision register 6
```

5.10.24 ELSE or | directive

The `ELSE` directive marks the beginning of a sequence of instructions and/or directives that are to be assembled if the preceding condition fails. | and `ELSE` are synonyms.

Syntax

```
ELSE
```

Usage

See *IF or [directive* on page 5-83.

5.10.25 END directive

The `END` directive informs the assembler that it has reached the end of a source file.

Syntax

```
END
```

Usage

Every assembly language source file must end with `END` on a line by itself.

If the source file has been included in a parent file by a `GET` directive, the assembler returns to the parent file and continues assembly at the first line following the `GET` directive. See *GET or INCLUDE directive* on page 5-82 for more information.

If `END` is reached in the top-level source file during the first pass without any errors, the second pass begins.

If `END` is reached in the top-level source file during the second pass, the assembler finishes the assembly and writes the appropriate output.

5.10.26 ENDFUNC directive

The `ENDFUNC` directive marks the end of an ATPCS-conforming function. `ENDP` and `ENDFUNC` are synonyms (see *FUNCTION directive* on page 5-78).

5.10.27 ENDIF or] directive

The `ENDIF` directive marks the end of a sequence of instructions and/or directives that are to be conditionally assembled. `]` and `ENDIF` are synonyms.

Syntax

```
ENDIF
```

Usage

See *IF or [directive* on page 5-83.

5.10.28 ENDP directive

The `ENDP` directive marks the end of an ATPCS-conforming function. `ENDP` and `ENDFUNC` are synonyms (see *FUNCTION directive* on page 5-78).

5.10.29 ENTRY directive

The ENTRY directive declares an entry point to a program.

Syntax

```
ENTRY
```

Usage

You must specify at least one ENTRY point for a program. If no ENTRY exists, a warning is generated at link time.

You must not use more than one ENTRY directive in a single source file. Not every source file has to have an ENTRY directive. If more than one ENTRY exists in a single source file, an error message is generated at assembly time.

Example

```
        AREA    ARMex, CODE, READONLY
        ENTRY                   ; Entry point for the application
```

5.10.30 EQU or * directive

The `EQU` directive gives a symbolic name to a numeric constant, a register-relative value or a program-relative value. `*` is a synonym for `EQU`.

Syntax

`name EQU expression`

where:

name is the symbolic name to assign to the value.

expression
is a fixed, register-relative, or program-relative value.

Usage

Use `EQU` to define constants. This is similar to the use of **`#define`** to define a constant in C.

See also:
- *Symbols* on page 5-30
- *Numeric constants* on page 5-31
- *MAP or ^ directive* on page 5-95
- *FIELD or # directive* on page 5-68
- *Register-relative and program-relative expressions* on page 5-119
- *Numeric expressions* on page 5-116
- *Numeric literals* on page 5-117.

Examples

```
abc     EQU     2       ; assigns the value 2 to the symbol abc.

xyz     EQU     label+8 ; assigns the value (label+8) to the
                        ; symbol xyz.
```

5.10.31 EXPORT or GLOBAL directive

The `EXPORT` directive declares a symbol that can be used by the linker to resolve symbol references in separate object and library files. `GLOBAL` is a synonym for `EXPORT`.

Syntax

`EXPORT` *`symbol`*`{[WEAK]}`

where:

`symbol`	is the symbol name to export. The symbol name is case-sensitive.
`[WEAK]`	means that this instance of *`symbol`* should only be imported into other sources if no other source exports an alternative instance.

Usage

Use `EXPORT` to allow code in other files to See symbols in the current file.

Use the `[WEAK]` attribute to inform the linker that a different instance of *`symbol`* should take precedence over this one, if a different one is available from another source.

See also *IMPORT directive* on page 5-84.

Example

```
        AREA    Example,CODE,READONLY
        EXPORT  DoAdd           ; Export the function name
                                ; to be used by external
                                ; modules.
DoAdd   ADD     r0,r0,r1
```

-

5.10.32 EXTERN directive

The `EXTERN` directive provides the assembler with a name that is not defined in the current assembly.

`EXTERN` is very similar to `IMPORT`, except that the name is not imported if no reference to it is found in the current assembly (see *IMPORT directive* on page 5-84, and *EXPORT or GLOBAL directive* on page 5-65).

Syntax

EXTERN *symbol*{[WEAK]}

where:

symbol	is a symbol name defined in a separately assembled source file, object file, or library. The symbol name is case-sensitive.
`[WEAK]`	prevents the linker generating an error message if the symbol is not defined elsewhere. It also prevents the linker searching libraries that are not already included.

Usage

The name is resolved at link time to a symbol defined in a separate object file. The symbol is treated as a program address. If `[WEAK]` is not specified, the linker generates an error if no corresponding symbol is found at link time.

If `[WEAK]` is specified and no corresponding symbol is found at link time:

- If the reference is the destination of a `B` or `BL` instruction, the value of the symbol is taken as the address of the following instruction. This makes the `B` or `BL` instruction effectively a `NOP`.
- Otherwise, the value of the symbol is taken as zero.

Example

This example tests to see if the C++ library has been linked, and branches conditionally on the result.

```
AREA    Example, CODE, READONLY
EXTERN  __CPP_INITIALIZE[WEAK]  ; If C++ library linked,
                                ; gets the address of
                                ; __CPP_INITIALIZE function.
LDR     r0,__CPP_INITIALIZE     ; If not linked, address
                                ; is zeroed.
CMP     r0,#0                   ; Test if zero.
BEQ     nocplusplus             ; Branch on the result.
```

5.10.33 FIELD or # directive

The `FIELD` directive describes space within a storage map that has been defined using the `MAP` directive. `#` is a synonym for `FIELD`.

Syntax

`{label} FIELD expression`

where:

label is an optional label. If specified, *label* is assigned the value of the storage location counter, `{VAR}`. The storage location counter is then incremented by the value of *expression*.

expression

is an expression that evaluates to the number of bytes to increment the storage counter.

Usage

If a storage map is set by a `MAP` directive that specifies a *base-register*, the base register is implicit in all labels defined by following `FIELD` directives, until the next `MAP` directive. These register-relative labels can be quoted in load and store instructions (see *MAP or ^ directive* on page 5-95).

Note

You must be careful when using `MAP`, `FIELD`, and register-relative labels. See the assembly language chapter in *ADS Developer Guide* for more information.

Example

The following example shows how register-relative labels are defined using the `MAP` and `FIELD` directives.

```
    MAP     0,r9         ; set {VAR} to the address stored in r9
    FIELD   4            ; increment {VAR} by 4 bytes
Lab FIELD   4            ; set Lab to the address [r9 + 4]
                         ; and then increment {VAR} by 4 bytes
    LDR     r0,Lab       ; equivalent to LDR r0,[r9,#4]
```

5.10.34 FN directive

The `FN` directive defines a name for a specified FPA floating-point register. The names f0-f7 and F0-F7 are predefined.

Syntax

`name` `FN` *`numeric-expression`*

where:

`name` is the name to be assigned to the floating-point register. *`name`* cannot be the same as any of the predefined names listed in *Predefined register and coprocessor names* on page 5-10.

`numeric-expression`

evaluates to a floating-point register number from 0 to 7.

Usage

Use `FN` to allocate convenient names to FPA floating-point registers, to help you to remember what you use each one for. Be careful to avoid conflicting uses of the same register under different names.

Example

```
energy  FN  6   ; defines energy as a symbol for
                ; floating-point register 6
```

5.10.35 FRAME ADDRESS directive

The `FRAME ADDRESS` directive describes how to calculate the canonical frame address for following instructions. You can only use it in functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives.

Syntax

`FRAME ADDRESS` *`reg`*`[,`*`offset`*`]`

where:

`reg`	is the register on which the canonical frame address is to be based. This is `sp` unless the function uses a separate frame pointer.
`offset`	is the offset of the canonical frame address from *`reg`*. If *`offset`* is zero, you may omit it.

Usage

Use `FRAME ADDRESS` if your code alters which register the canonical frame address is based on, or if it alters the offset of the canonical frame address from the register. You must use `FRAME ADDRESS` immediately after the instruction which changes the calculation of the canonical frame address.

Note

If your code uses a single instruction to save registers and alter the stack pointer, you can use `FRAME PUSH` instead of using both `FRAME ADDRESS` and `FRAME SAVE` (see *FRAME PUSH directive* on page 5-72).

If your code uses a single instruction to load registers and alter the stack pointer, you can use `FRAME POP` instead of using both `FRAME ADDRESS` and `FRAME RESTORE` (see *FRAME POP directive* on page 5-71).

Example

```
_fn     FUNCTION        ; CFA (Canonical Frame Address) is value
                        ; of sp on entry to function
        STMFD   sp!, {r4,fp,ip,lr,pc}
        FRAME PUSH {r4,fp,ip,lr,pc}
        SUB     sp,sp,#4            ; CFA offset now changed
        FRAME ADDRESS sp,24         ; - so we correct it
        ADD     fp,sp,#20
        FRAME ADDRESS fp,4          ; New base register
        ; code using fp to base call-frame on, instead of sp
```

5.10.36 FRAME POP directive

Use the `FRAME POP` directive to inform the assembler when the callee reloads registers. You can only use it within functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives.

You need not do this after the last instruction in a function.

Syntax

```
FRAME POP {reg | reglist}
```

where:

reg	is the one register restored to the value it had on entry to the function.
reglist	is a list of registers restored to the values they had on entry to the function.

Usage

`FRAME POP` is equivalent to a `FRAME ADDRESS` and a `FRAME RESTORE` directive. You may use it when a single instruction loads registers and alters the stack pointer.

You must use `FRAME POP` immediately after the instruction it refers to.

The assembler calculates the new offset for the canonical frame address. It assumes that:

- each ARM register popped occupied 4 bytes on the stack
- each FPA floating-point register popped occupied 12 bytes on the stack
- each VFP single-precision register popped occupied 4 bytes on the stack, plus an extra 4-byte word for each list.

See *FRAME ADDRESS directive* on page 5-70 and *FRAME RESTORE directive* on page 5-74.

5.10.37 FRAME PUSH directive

Use the `FRAME PUSH` directive to inform the assembler when the callee saves registers, normally at function entry. You can only use it within functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives.

Syntax

FRAME PUSH {*reg* | *reglist*}

where:

reg	is a register stored immediately below the canonical frame address.
reglist	is a list of registers stored consecutively below the canonical frame address.

Usage

`FRAME PUSH` is equivalent to a `FRAME ADDRESS` and a `FRAME SAVE` directive. You may use it when a single instruction saves registers and alters the stack pointer.

You must use `FRAME PUSH` immediately after the instruction it refers to.

The assembler calculates the new offset for the canonical frame address. It assumes that:

- each ARM register pushed occupies 4 bytes on the stack
- each FPA floating-point register pushed occupies 12 bytes on the stack
- each VFP single-precision register pushed occupies 4 bytes on the stack, plus an extra 4-byte word for each list.

See *FRAME ADDRESS directive* on page 5-70 and *FRAME SAVE directive* on page 5-75.

Example

```
p    PROC ; Canonical frame address is sp + 0
     EXPORT  p
     STMFD   sp!,{r4-r6,lr}
          ; sp has moved relative to the canonical frame address,
          ; and registers r4, r5, r6 and lr are now on the stack
     FRAME PUSH {r4-r6,lr}
          ; Equivalent to:
          ; FRAME ADDRESS    sp,16       ; 16 bytes in {r4-r6,lr}
          ; FRAME SAVE    {r4-r6,lr},-16
```

5.10.38 FRAME REGISTER directive

Use the `FRAME REGISTER` directive to maintain a record of the locations of function arguments held in registers. You can only use it within functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives.

Syntax

```
FRAME REGISTER reg1,reg2
```

where:

reg1	is the register that held the argument on entry to the function.
reg2	is the register in which the value is preserved.

Usage

Use the `FRAME REGISTER` directive when you use a register to preserve an argument that was held in a different register on entry to a function.

5.10.39 FRAME RESTORE directive

Use the `FRAME RESTORE` directive to inform the assembler that the contents of specified registers have been restored to the values they had on entry to the function. You can only use it within functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives.

Syntax

```
FRAME RESTORE {reg | reglist}
```

where:

reg	is the one register whose contents have been restored.
reglist	is a list of registers whose contents have been restored.

Usage

Use `FRAME RESTORE` immediately after the callee reloads registers from the stack. You need not do this after the last instruction in a function.

reglist may contain integer registers or floating-point registers, but not both.

Note

If your code uses a single instruction to load registers and alter the stack pointer, you can use `FRAME POP` instead of using both `FRAME RESTORE` and `FRAME ADDRESS` (see *FRAME POP directive* on page 5-71).

5.10.40 FRAME SAVE directive

The `FRAME SAVE` directive describes the location of saved register contents relative to the canonical frame address. You can only use it within functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives.

Syntax

`FRAME SAVE {`*`reg`*` | `*`reglist`*`}, `*`offset`*

where:

`reg`	is the one register stored at *`offset`* from the canonical frame address.
`reglist`	is a list of registers stored consecutively starting at *`offset`* from the canonical frame address.

Usage

Use `FRAME SAVE` immediately after the callee stores registers onto the stack.

`reglist` may include registers which are not required for backtracing. The assembler determines which registers it needs to record in the DWARF call frame information.

Note

If your code uses a single instruction to save registers and alter the stack pointer, you can use `FRAME PUSH` instead of using both `FRAME SAVE` and `FRAME ADDRESS` (see *FRAME PUSH directive* on page 5-72).

5.10.41 FRAME STATE REMEMBER directive

The `FRAME STATE REMEMBER` directive saves the current information on how to calculate the canonical frame address and locations of saved register values. You can only use it within functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives.

Syntax

```
FRAME STATE REMEMBER
```

Usage

During an inline exit sequence the information about calculation of canonical frame address and locations of saved register values may change. After the exit sequence another branch may continue using the same information as before. Use `FRAME STATE REMEMBER` to preserve this information, and `FRAME STATE RESTORE` to restore it.

These directives may be nested. Each `FRAME STATE RESTORE` directive must have a corresponding `FRAME STATE REMEMBER` directive. See:

- *FRAME STATE RESTORE directive* on page 5-77
- *FUNCTION directive* on page 5-78.

Example

```
        ; function code
        FRAME STATE REMEMBER
            ; save frame state before in-line exit sequence
        LDMFD   sp!,{r4-r6,pc}
            ; no need to FRAME POP here, as control has
            ; transferred out of the function
        FRAME STATE RESTORE
            ; end of exit sequence, so restore state
exitB   ; code for exitB
        LDMFD   sp!,{r4-r6,pc}
        ENDP
```

5.10.42 FRAME STATE RESTORE directive

The `FRAME STATE RESTORE` directive restores information about how to calculate the canonical frame address and locations of saved register values. You can only use it within functions with `FUNCTION` and `ENDFUNC` or `PROC` and `ENDP` directives.

Syntax

```
FRAME STATE RESTORE
```

Usage

See:

- *FRAME STATE REMEMBER directive* on page 5-76
- *FUNCTION directive* on page 5-78.

5.10.43 FUNCTION directive

The `FUNCTION` directive marks the start of an ATPCS-conforming function. `FUNCTION` and `PROC` are synonyms.

Syntax

`label` `FUNCTION`

Usage

`FUNCTION`:

- helps you to avoid errors in function construction, particularly when you are modifying existing code
- allows the assembler to alert you to errors in function construction
- enables backtracing of function calls during debugging.

Use `FUNCTION` to mark the start of functions. The assembler uses `FUNCTION` to identify the start of a function when producing DWARF call frame information for ELF.

`FUNCTION` sets the canonical frame address to be `SP`, and the frame state stack to be empty.

Each `FUNCTION` directive must have a matching `ENDFUNC` directive. You must not nest `FUNCTION`/`ENDFUNC` pairs, and they must not contain `PROC` or `ENDP` directives.

See the assembly language chapter in *ADS Developer Guide* for information about the usage of `FUNCTION`.

See also *FRAME ADDRESS directive* on page 5-70 to *FRAME STATE RESTORE directive* on page 5-77.

Example

```
dadd    FUNCTION
        EXPORT  dadd
        STMFD   sp!,{r4-r6,lr}
        FRAME PUSH {r4-r6,lr}
        ; subroutine body
        LDMFD   sp!,{r4-r6,pc}
        ENDFUNC
```

5.10.44 GBLA directive

The `GBLA` directive declares and initializes a global arithmetic variable. The range of values that arithmetic variables may take is the same as that of numeric expressions (see *Numeric expressions* on page 5-116).

Syntax

`GBLA` *`variable-name`*

where:

`variable-name`

is the name of the arithmetic variable. *`variable-name`* must be unique amongst symbols within a source file.

`variable-name` is initialized to 0.

Usage

Using `GBLA` for a variable that is already defined re-initializes the variable to 0. The scope of the variable is limited to the source file that contains it.

Set the value of the variable with the `SETA` directive (see *SETA directive* on page 5-103).

See *LCLA directive* on page 5-88 for information on setting local arithmetic variables.

Global variables can also be set with the `-predefine` assembler command-line option. See *Command syntax* on page 5-4 for more information.

Example

```
            GBLA    objectsize    ; declare the variable name
objectsize  SETA    0xff          ; set its value
            SPACE   objectsize    ; quote the variable
```

-

5.10.45 GBLL directive

The `GBLL` directive declares and initializes a global logical variable. Possible values of a logical variable are `{TRUE}` and `{FALSE}`.

Syntax

`GBLL` *`variable-name`*

where:

`variable-name`

is the name of the logical variable. *`variable-name`* must be unique amongst symbols within a source file.

`variable-name` is initialized to `{FALSE}`.

Usage

Using `GBLL` for a variable that is already defined re-initializes the variable to `{FALSE}`. The scope of the variable is limited to the source file that contains it.

Set the value of the variable with the `SETL` directive (see *SETL directive* on page 5-104).

See *LCLL directive* on page 5-89 for information on setting local logical variables.

Global variables can also be set with the `-predefine` assembler command-line option. See *Command syntax* on page 5-4 for more information.

Example

```
        GBLL    testrun
testrun SETL    {TRUE}

        IF      testrun
        ; testcode
        ENDIF
```

5.10.46 GBLS directive

The `GBLS` directive declares and initializes a global string variable. The range of values that string variables may take is the same as that of string expressions (see *String expressions* on page 5-115).

Syntax

`GBLS` *`variable-name`*

where:

`variable-name`

is the name of the string variable. *`variable-name`* must be unique amongst symbols within a source file.

`variable-name` is initialized to a null string, `""`.

Usage

Using `GBLS` for a variable that is already defined re-initializes the variable to a null string. The scope of the variable is limited to the source file that contains it.

Set the value of the variable with the `SETS` directive (see *SETS directive* on page 5-105).

See *LCLS directive* on page 5-90 for information on setting local string variables.

Global variables can also be set with the `-predefine` assembler command-line option. See *Command syntax* on page 5-4 for more information.

Example

```
        GBLS    version          ; declare the variable
version SETS    "Version 1.0"    ; set its value
        ; code
        INFO    0,version        ; use the variable
```

-

5.10.47 GET or INCLUDE directive

The `GET` directive includes a file within the file being assembled. The included file is assembled at the location of the `GET` directive. `INCLUDE` is a synonym for `GET`.

Syntax

`GET` *`filename`*

where:

`filename`	is the name of the file to be included in the assembly. The assembler accepts pathnames in either UNIX or MS-DOS format.

Usage

`GET` is useful for including macro definitions, `EQU`s, and storage maps in an assembly. When assembly of the included file is complete, assembly continues at the line following the `GET` directive.

By default the assembler searches the current place for included files. The current place is the directory where the calling file is located. Use the `-i` assembler command-line option to add directories to the search path. File names and directory names containing spaces must be enclosed in double quotes (" ").

The included file may contain additional GET directives to include other files (see *Nesting directives* on page 5-36).

If the included file is in a different directory from the current place, this becomes the current place until the end of the included file. The previous current place is then restored.

`GET` cannot be used to include object files (see *INCBIN directive* on page 5-85).

Example

```
    AREA    Example, CODE, READONLY
    GET     file1.s               ; includes file1 if it exists
                                  ; in the current place.
    GET     c:\project\file2.s    ; includes file2
    GET     c:\Program files\file3.s  ; space is allowed
```

5.10.48 GLOBAL directive

See *EXPORT or GLOBAL directive* on page 5-65

5.10.49 IF or [directive

The `IF` directive introduces a condition that is used to decide whether to assemble a sequence of instructions and/or directives. `[` and `IF` are synonyms.

Syntax

```
IF logical-expression

    ...

{ELSE

    ...}

ENDIF
```

where:

`logical-expression`
is an expression that evaluates to either `{TRUE}` or `{FALSE}`.

See *Relational operators* on page 5-123.

Usage

Use `IF` with `ENDIF`, and optionally with `ELSE`, for sequences of instructions and/or directives that are only to be assembled or acted on under a specified condition (see also *ELSE or | directive* on page 5-61 and *ENDIF or] directive* on page 5-62).

`IF...ENDIF` conditions can be nested (see *Nesting directives* on page 5-36).

Example

```
        IF Version = "1.0"
            ; code and/or
            ; directives
        ELSE
            ; code and/or
            ; directives
        ENDIF
```

5.10.50 IMPORT directive

The `IMPORT` directive provides the assembler with a name that is not defined in the current assembly.

`IMPORT` is very similar to `EXTERN`, except that the name is imported whether or not it is referred to in the current assembly (see *EXTERN directive* on page 5-66, and *EXPORT or GLOBAL directive* on page 5-65).

Syntax

```
IMPORT symbol{[WEAK]}
```

where:

`symbol`	is a symbol name defined in a separately assembled source file, object file, or library. The symbol name is case-sensitive.
`WEAK`	prevents the linker generating an error message if the symbol is not defined elsewhere. It also prevents the linker searching libraries that are not already included.

Usage

The name is resolved at link time to a symbol defined in a separate object file. The symbol is treated as a program address. If `[WEAK]` is not specified, the linker generates an error if no corresponding symbol is found at link time.

If `[WEAK]` is specified and no corresponding symbol is found at link time:

- If the reference is the destination of a `B` or `BL` instruction, the value of the symbol is taken as the address of the following instruction. This makes the `B` or `BL` instruction effectively a `NOP`.
- Otherwise, the value of the symbol is taken as zero.

To avoid trying to access symbols that are not found at link time, use code like the example in *EXTERN directive* on page 5-66.

5.10.51 INCBIN directive

The `INCBIN` directive includes a file within the file being assembled. The file is included as it is, without being assembled.

Syntax

`INCBIN` *`filename`*

where:

`filename`	is the name of the file to be included in the assembly. The assembler accepts pathnames in either UNIX or MS-DOS format.

Usage

You can use `INCBIN` to include executable files, literals, or any arbitrary data. The contents of the file are added to the current ELF section, byte for byte, without being interpreted in any way. Assembly continues at the line following the `INCBIN` directive.

By default the assembler searches the current place for included files. See *GET or INCLUDE directive* on page 5-82 for information on the current place. Use the `-i` assembler command-line option to add directories to the search path.

File names and directory names must not contain spaces.

Example

```
        AREA    Example, CODE, READONLY
        INCBIN  file1.dat                ; includes file1 if it
                                         ; exists in the
                                         ; current place.
        INCBIN  c:\project\file2.txt     ; includes file2
```

5.10.52 INCLUDE directive

See *GET or INCLUDE directive* on page 5-82

5.10.53 INFO or ! directive

The `INFO` directive supports diagnostic generation on either pass of the assembly.

! is a synonym for `INFO`.

Syntax

`INFO` *`numeric-expression, string-expression`*

where:

`numeric-expression`

is a numeric expression that is evaluated during assembly. If the expression evaluates to zero:

- no action is taken during pass one
- *`string-expression`* is printed during pass two.

If the expression does not evaluate to zero, *`string-expression`* is printed as an error message and the assembly fails.

`string-expression`

is an expression that evaluates to a string.

Usage

`INFO` provides a flexible means for creating custom error messages. See *Numeric expressions* on page 5-116 and *String expressions* on page 5-115 for additional information on numeric and string expressions.

See also *ASSERT directive* on page 5-41.

Examples

```
        INFO    0, "Version 1.0"

        IF endofdata <= label1
            INFO    4, "Data overrun at label1"
        ENDIF
```

5.10.54 KEEP directive

The `KEEP` directive instructs the assembler to retain local symbols in the symbol table in the object file.

Syntax

`KEEP {`*`symbol`*`}`

where:

`symbol`	is the name of the local symbol to keep. If *`symbol`* is not specified, all local symbols are kept except register-relative symbols.

Usage

By default, the only symbols that the assembler describes in its output object file are:

- exported symbols
- symbols that are relocated against.

Use `KEEP` to preserve local symbols that can be used to help debugging. Kept symbols appear in the ARM debuggers and in linker map files.

`KEEP` cannot preserve register-relative symbols (see *MAP or ^ directive* on page 5-95).

Example

```
label   ADC     r2,r3,r4
        KEEP    label       ; makes label available to debuggers
        ADD     r2,r2,r5
```

5.10.55 LCLA directive

The `LCLA` directive declares and initializes a local arithmetic variable. Local variables can be declared only within a macro.

The range of values that arithmetic variables may take is the same as that of numeric expressions (see *Numeric expressions* on page 5-116).

Syntax

`LCLA` *`variable-name`*

where:

`variable-name`

is the name of the variable to set. The name must be unique within the macro that contains it. The initial value of the variable is 0.

Usage

The scope of the variable is limited to a particular instantiation of the macro that contains it (see *MACRO directive* on page 5-92).

Using `LCLA` for a variable that is already defined re-initializes the variable to 0.

Set the value of the variable with the `SETA` directive (see *SETA directive* on page 5-103).

See *GBLA directive* on page 5-79 for information on declaring global arithmetic variables.

Example

```
                                ; Calculate the next-power-of-2
                                ; number >= the value given.
        MACRO                   ; Declare a macro
$rslt   NPOW2   $value          ; Macro prototype line
        LCLA    newval          ; Declare local arithmetic
                                ; variable newval.
newval  SETA    1               ; Set value of newval to 1
        WHILE   (newval < $value)
                                ; Repeat a loop that
newval  SETA    (newval :SHL:1) ; multiplies newval by 2
        WEND                    ; until newval >= $value.
$rslt   EQU     (newval)        ; Return newval in $rslt
        MEND                    ; No runtime instructions here!
```

5.10.56 LCLL directive

The `LCLL` directive declares and initializes a local logical variable. Local variables can be declared only within a macro. Possible values of a logical variable are `{TRUE}` and `{FALSE}`.

Syntax

`LCLL` *`variable-name`*

where:

`variable-name`

is the name of the variable to set. The name must be unique within the macro that contains it. The initial value of the variable is `{FALSE}`.

Usage

The scope of the variable is limited to a particular instantiation of the macro that contains it (see *MACRO directive* on page 5-92).

Using `LCLL` for a variable that is already defined re-initializes the variable to `{FALSE}`.

Set the value of the variable with the `SETL` directive (see *SETL directive* on page 5-104).

See *GBLL directive* on page 5-80 for information on declaring global logical variables.

Example

```
        MACRO                    ; Declare a macro
$label  cases   $x               ; Macro prototype line
        LCLL    xisodd           ; Declare local logical variable
                                 ; xisodd.
xisodd  SETL    $x:MOD:2=1       ; Set value of xisodd according
                                 ; to $x
$label  ; code
        IF      xisodd           ; Assemble following code only
                                 ; if $x is odd.
        ; code
        ENDIF
        MEND                     ; End of macro
```

5.10.57 LCLS directive

The `LCLS` directive declares and initializes a local string variable. Local variables can be declared only within a macro. The initial value of the variable is a null string, `""`.

Syntax

`LCLS` *`variable-name`*

where:

`variable-name`
is the name of the variable to set. The name must be unique within the macro that contains it.

Usage

The scope of the variable is limited to a particular instantiation of the macro that contains it (see *MACRO directive* on page 5-92).

Using `LCLS` for a variable that is already defined re-initializes the variable to a null string.

Set the value of the variable with the `SETS` directive (see *SETS directive* on page 5-105).

See *GBLS directive* on page 5-81 for information on declaring global logical variables.

Example

```
        MACRO                           ; Declare a macro
$label  message $a                      ; Macro prototype line
        LCLS    err                     ; Declare local string
                                        ; variable err.
err     SETS    "error no: "            ; Set value of err
$label  ; code
        INFO    0, "err":CC::STR:$a     ; Use string
        MEND
```

5.10.58 LTORG directive

The `LTORG` directive instructs the assembler to assemble the current literal pool immediately.

Syntax

```
LTORG
```

Usage

The assembler assembles the current literal pool at the end of every code section. The end of a code section is determined by the `AREA` directive at the beginning of the following section, or the end of the assembly.

These default literal pools may sometimes not be within range of some `LDR`, `LDFD`, and `LDFS` pseudo-instructions. See *LDR ARM pseudo-instruction* on page 5-21 and *LDR Thumb pseudo-instruction* on page 5-26 for more information. Use `LTORG` to ensure that a literal pool is assembled within range. Large programs may require several literal pools.

Place `LTORG` directives after unconditional branches or subroutine return instructions so that the processor does not attempt to execute the constants as instructions.

The assembler word-aligns data in literal pools.

Example

```
        AREA    Example, CODE, READONLY
start   BL      func1

func1                              ; function body
        ; code
        LDR     r1,=0x55555555     ; => LDR R1, [pc, #offset to
                                   ; Literal Pool 1]
        ; code
        MOV     pc,lr              ; end function
        LTORG                      ; Literal Pool 1 contains
                                   ; literal &55555555.

data    SPACE   4200               ; Clears 4200 bytes of memory,
                                   ; starting at current location.
        END                        ; Default literal pool is empty.
```

5.10.59 MACRO directive

The `MACRO` directive marks the start of the definition of a macro. Macro expansion terminates at the `MEND` directive. See the assembly language chapter in *ADS Developer Guide* for further information.

Syntax

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

```
          MACRO
{$label}  macroname {$parameter{,$parameter}...}
          ; code
          MEND
```

where:

`$label`	is a parameter that is substituted with a symbol given when the macro is invoked. The symbol is usually a label.
`macroname`	is the name of the macro. It must not begin with an instruction or directive name.
`$parameter`	is a parameter that is substituted when the macro is invoked. A default value for a parameter may be set using this format: *`$parameter="default value"`* Double quotes must be used if there are any spaces within, or at either end of, the default value.

Usage

There must be no unclosed `WHILE...WEND` loops or unclosed `IF...ENDIF` conditions when the `MEND` directive is reached. See *MEXIT directive* on page 5-96 if you need to allow an early exit from a macro, for example from within a loop.

Within the macro body, parameters such as *`$label`*, *`$parameter`* can be used in the same way as other variables (see *Assembly time substitution of variables* on page 5-32). They are given new values each time the macro is invoked. Parameters must begin with `$` to distinguish them from ordinary symbols. Any number of parameters can be used.

`$label` is optional. It is useful if the macro defines internal labels. It is treated as a parameter to the macro. It does not necessarily represent the first instruction in the macro expansion. The macro defines the locations of any labels.

Use `|` as the argument to use the default value of a parameter. An empty string is used if the argument is omitted.

In a macro that uses several internal labels, it is useful to define each internal label as the base label with a different suffix.

Use a dot between a parameter and following text, or a following parameter, if a space is not required in the expansion. Do not use a dot between preceding text and a parameter.

Macros define the scope of local variables (see *LCLA directive* on page 5-88, *LCLL directive* on page 5-89, and *LCLS directive* on page 5-90).

Macros can be nested (see *Nesting directives* on page 5-36).

Examples

Example 5-1 Macro definition and invocation

```
                MACRO                           ; start macro definition
$label          xmac    $p1,$p2
                ; code
$label.loop1    ; code
                ; code
                BGE     $label.loop1
$label.loop2    ; code
                BL      $p1
                BGT     $label.loop2
                ; code
                ADR     $p2
                ; code
                MEND                            ; end macro definition

 ; macro invocation

abc             xmac    subr1,de        ; invoke macro
                ; code                  ; this is what is
abcloop1        ; code                  ; is produced when
                ; code                  ; the xmac macro is
                BGE     abcloop1        ; expanded
abcloop2        ; code
                BL      subr1
                BGT     abcloop2
                ; code
                ADR     de
                ; code
```

Example 5-2 Macro default parameters

```
        MACRO                            ; Macro definition
        diagnose  $param1="default"      ; This macro produces
        INFO      0,"$param1"            ; assembly-time diagnostics
        MEND                             ; (on second assembly pass)

 ; macro expansion

        diagnose            ; Prints blank line at assembly-time
        diagnose "hello"    ; Prints "hello" at assembly-time
        diagnose |          ; Prints "default" at assembly-time
```

5.10.60 MAP or ^ directive

The `MAP` directive sets the origin of a storage map to a specified address. The storage-map location counter, `{VAR}`, is set to the same address. `^` is a synonym for `MAP`.

Syntax

`MAP` *`expression{,base-register}`*

where:

`expression`

is a numeric or program-relative expression:

- If *`base-register`* is not specified, *`expression`* evaluates to the address where the storage map starts. The storage map location counter is set to this address.
- If the expression is program-relative, you must have defined the label before you use it in the map. The map requires the definition of the label during the first pass of the assembler.

`base-register`

specifies a register. If *`base-register`* is specified, the address where the storage map starts is the sum of *`expression`*, and the value in *`base-register`* at runtime.

Usage

Use the `MAP` directive in combination with the `FIELD` directive to describe a storage map.

Specify *`base-register`* to define register-relative labels. The base register becomes implicit in all labels defined by following `FIELD` directives, until the next `MAP` directive. The register-relative labels can be used in load and store instructions. See *FIELD or # directive* on page 5-68 for an example.

The `MAP` directive can be used any number of times to define multiple storage maps.

The `{VAR}` counter is set to zero before the first `MAP` directive is used.

Examples

```
        MAP     0,r9
        MAP     0xff,r9
```

5.10.61 MEND directive

The `MEND` directive marks the end of a macro definition (see *MACRO directive* on page 5-92).

5.10.62 MEXIT directive

The `MEXIT` directive is used to exit a macro definition before the end.

Syntax

```
MEXIT
```

Usage

Use `MEXIT` when you need an exit from within the body of a macro. Any unclosed `WHILE...WEND` loops or `IF...ENDIF` conditions within the body of the macro are closed by the assembler before the macro is exited.

See also *MACRO directive* on page 5-92.

Example

```
        MACRO
$abc    macro   abc     $param1,$param2
        ; code
        WHILE condition1
            ; code
            IF condition2
                ; code
                MEXIT
            ELSE
                ; code
            ENDIF
        WEND
        ; code
        MEND
```

5.10.63 NOFP directive

The `NOFP` directive disallows floating-point instructions in an assembly language source file.

Syntax

```
NOFP
```

Usage

Use `NOFP` to ensure that no floating-point instructions are used in situations where there is no support for floating-point instructions either in software or in target hardware.

If a floating-point instruction occurs after the `NOFP` directive, an `Unknown opcode` error is generated and the assembly fails.

If a `NOFP` directive occurs after a floating-point instruction, the assembler generates the error:

```
Too late to ban floating point instructions
```

and the assembly fails.

5.10.64 OPT directive

The `OPT` directive sets listing options from within the source code.

Syntax

`OPT n`

where:

n is the `OPT` directive setting. Table 5-9 lists valid settings.

Table 5-9 OPT directive settings

OPT n	Effect
1	Turns on normal listing.
2	Turns off normal listing.
4	Page throw. Issues an immediate form feed and starts a new page.
8	Resets the line number counter to zero.
16	Turns on listing for `SET`, `GBL` and `LCL` directives.
32	Turns off listing for `SET`, `GBL` and `LCL` directives.
64	Turns on listing of macro expansions.
128	Turns off listing of macro expansions.
256	Turns on listing of macro invocations.
512	Turns off listing of macro invocations.
1024	Turns on the first pass listing.
2048	Turns off the first pass listing.
4096	Turns on listing of conditional directives.
8192	Turns off listing of conditional directives.
16384	Turns on listing of `MEND` directives.
32768	Turns off listing of `MEND` directives.

Usage

Specify the `-list` assembler option to turn on listing.

-

By default the `-list` option produces a normal listing that includes variable declarations, macro expansions, call-conditioned directives, and `MEND` directives. The listing is produced on the second pass only. Use the `OPT` directive to modify the default listing options from within your code. See *Command syntax* on page 5-4 for information on the `-list` option.

You can use `OPT` to format code listings. For example, you can specify a new page before functions and sections.

Example

```
        AREA    Example, CODE, READONLY
start   ; code
        ; code
        BL      func1
        ; code
        OPT 4                   ; places a page break before func1
func1   ; code
```

5.10.65 PROC directive

The `PROC` directive marks the start of an ATPCS-conforming function. `PROC` and `FUNCTION` are synonyms (see *FUNCTION directive* on page 5-78).

5.10.66 REQUIRE directive

The `REQUIRE` directive specifies a dependency between sections.

Syntax

`REQUIRE` *`label`*

where:

`label`	is the name of the required label.

Usage

Use `REQUIRE` to ensure that a related section is included, even if it is not directly called. If the section containing the `REQUIRE` directive is included in a link, the linker also includes the section containing the definition of the specified label.

5.10.67 RLIST directive

The `RLIST` (register list) directive gives a name to a set of registers.

Syntax

```
name RLIST {list-of-registers}
```

where:

`name` is the name to be given to the set of registers.

`list-of-registers`
is a comma-delimited list of register names and/or register ranges. The register list must be enclosed in braces.

Usage

Use `RLIST` to give a name to a set of registers to be transferred by the `LDM` or `STM` instructions.

`LDM` and `STM` always put the lowest physical register numbers at the lowest address in memory, regardless of the order they are supplied to the `LDM` or `STM` instruction. If you have defined your own symbolic register names it can be less apparent that a register list is not in increasing register order.

Use the `-checkreglist` assembler option to ensure that the registers in a register list are supplied in increasing register order. If registers are not supplied in increasing register order, a warning is issued.

Example

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

5.10.68 RN directive

The `RN` directive defines a register name for a specified register.

Syntax

`name` `RN` *`numeric-expression`*

where:

`name` is the name to be assigned to the register. *`name`* cannot be the same as any of the predefined names listed in *Predefined register and coprocessor names* on page 5-10.

`numeric-expression`
evaluates to a register number from 0 to 15.

Usage

Use `RN` to allocate convenient names to registers, to help you to remember what you use each register for. Be careful to avoid conflicting uses of the same register under different names.

Examples

```
regname     RN  11  ; defines regname for register 11

sqr4        RN  r6  ; defines sqr4 for register 6
```

5.10.69 ROUT directive

The `ROUT` directive marks the boundaries of the scope of local labels (see *Local labels* on page 5-34).

Syntax

`{name} ROUT`

where:

name	is the name to be assigned to the scope.

Usage

Use the `ROUT` directive to limit the scope of local labels. This makes it easier for you to avoid referring to a wrong label by accident. The scope of local labels is the whole area if there are no `ROUT` directives in it (see *AREA directive* on page 5-39).

Use the *name* option to ensure that each reference is to the correct local label. If the name of a label or a reference to a label does not match the preceding `ROUT` directive, the assembler generates an error message and the assembly fails.

Example

```
            ; code
routineaA   ROUT            ; ROUT is not necessarily a routine
            ; code
3routineA   ; code          ; this label is checked
            ; code
            BEQ     %4routineA   ; this reference is checked
            ; code
            BGE     %3      ; refers to 3 above, but not checked
            ; code
4routineA   ; code          ; this label is checked
            ; code
otherstuff  ROUT            ; start of next scope
```

5.10.70 SETA directive

The `SETA` directive sets the value of a local or global arithmetic variable.

Syntax

`variable-name` `SETA` *`expression`*

where:

`variable-name`

is the name of a variable declared by a `GBLA` or `LCLA` directive.

`expression`

is a numeric expression (see *Numeric expressions* on page 5-116).

Usage

You must declare *`variable-name`* using a `GBLA` or `LCLA` directive before using `SETA`. See *GBLA directive* on page 5-79 and *LCLA directive* on page 5-88 for more information.

You can also predefine variable names on the command line. See *Command syntax* on page 5-4 for more information.

Example

```
                GBLA    VersionNumber
VersionNumber   SETA    21
```

5.10.71 SETL directive

The `SETL` directive sets the value of a local or global logical variable.

Syntax

`variable-name` `SETL` *`expression`*

where:

`variable-name`

is the name of a variable declared by a `GBLL` or `LCLL` directive.

`expression`

is an expression that evaluates to either `{TRUE}` or `{FALSE}`.

Usage

You must declare *`variable-name`* using a `GBLL` or `LCLL` directive before using `SETL`. See *GBLL directive* on page 5-80 and *LCLL directive* on page 5-89 for more information.

You can also predefine variable names on the command line. See *Command syntax* on page 5-4 for more information.

Example

```
        GBLL    Debug
Debug   SETL    {TRUE}
```

5.10.72 SETS directive

The `SETS` directive sets the value of a local or global string variable.

Syntax

`variable-name` `SETS` *`string-expression`*

where:

`variable-name`

is the name of the variable declared by a `GBLS` or `LCLS` directive.

`string-expression`

is a string expression (see *String expressions* on page 5-115).

Usage

You must declare *`variable-name`* using a `GBLS` or `LCLS` directive before using `SETS`. See *GBLS directive* on page 5-81 and *LCLS directive* on page 5-90 for more information.

You can also predefine variable names on the command line. See *Command syntax* on page 5-4 for more information.

Example

```
                GBLS    VersionString
VersionString   SETS    "Version 1.0"
```

5.10.73 SN directive

The SN directive defines a name for a specified single-precision VFP register. The names s0-s31 and S0-S31 are predefined.

Syntax

name SN *numeric-expression*

where:

name	is the name to be assigned to the VFP register. *name* cannot be the same as any of the predefined names listed in *Predefined register and coprocessor names* on page 5-10.
numeric-expression	evaluates to a single-precision VFP register number from 0 to 31.

Usage

Use SN to allocate convenient names to single-precision VFP registers, to help you to remember what you use each one for. Be careful to avoid conflicting uses of the same register under different names.

You cannot specify a vector length in an SN directive (see *VFP directives and notation* on page 5-11).

See also *DN directive* on page 5-60.

Example

```
energy  SN  16  ; defines energy as a symbol for
                ; VFP single-precision register 16
```

5.10.74 SPACE or % directive

The `SPACE` directive reserves a zeroed block of memory. `%` is a synonym for `SPACE`.

Syntax

{*label*} SPACE *numeric-expression*

where:

numeric-expression

evaluates to the number of zeroed bytes to reserve (see *Numeric expressions* on page 5-116).

Usage

You *must* use a `DATA` directive if you use `SPACE` to define labeled data within Thumb code. See *DATA directive* on page 5-46 for more information.

Use the `ALIGN` directive to align any code following a `SPACE` directive. See *ALIGN directive* on page 5-37 for more information.

See also:

- *DCB or = directive* on page 5-47
- *DCD or & directive* on page 5-48
- *DCDO directive* on page 5-49
- *DCDU directive* on page 5-50
- *DCW directive* on page 5-58
- *DCWU directive* on page 5-59.

Example

```
        AREA    MyData, DATA, READWRITE
data1   SPACE   255     ; defines 255 bytes of zeroed store
```

-

5.10.75 SUBT directive

The `SUBT` directive places a subtitle on the pages of a listing file. The subtitle is printed on each page until a new `SUBT` directive is issued.

Syntax

`SUBT` *`subtitle`*

where:

`subtitle` is the subtitle.

Usage

Use `SUBT` to place a subtitle at the top of the pages of a listing file. Subtitles appear in the line below the titles (see *TTL directive* on page 5-109). If you want the subtitle to appear on the first page, the `SUBT` directive must be on the first line of the source file.

Use additional `SUBT` directives to change subtitles. Each new `SUBT` directive takes effect from the top of the next page.

Example

```
        TTL     First Title     ; places a title on the first
                                ; and subsequent pages of a
                                ; listing file.
        SUBT    First Subtitle  ; places a subtitle on the
                                ; second and subsequent pages
                                ; of a listing file.
```

5.10.76 TTL directive

The `TTL` directive inserts a title at the start of each page of a listing file. The title is printed on each page until a new `TTL` directive is issued.

Syntax

`TTL` *`title`*

where:

`title`	is the title.

Usage

Use the `TTL` directive to place a title at the top of the pages of a listing file. If you want the title to appear on the first page, the `TTL` directive must be on the first line of the source file.

Use additional `TTL` directives to change the title. Each new `TTL` directive takes effect from the top of the next page.

Example

```
        TTL     First Title     ; places a title on the first
                                ; and subsequent pages of a
                                ; listing file.
```

-

5.10.77 VFPASSERT SCALAR

The `VFPASSERT SCALAR` directive informs the assembler that following VFP instructions are in scalar mode.

Syntax

```
VFPASSERT SCALAR
```

Usage

Use the `VFPASSERT SCALAR` directive to mark the end of any block of code where the VFP mode is `VECTOR`.

Place the `VFPASSERT SCALAR` directive immediately after the instruction that makes the change. This is usually an `FMXR` instruction, but could be a `BL` instruction.

If a function expects the VFP to be in vector mode on exit, place a `VFPASSERT SCALAR` directive immediately after the last instruction. Such a function would not be ATPCS conformant. See the ATPCS chapter in *ADS Developer Guide* for further information.

See:

- *VFP directives and notation* on page 5-11
- *VFPASSERT VECTOR* on page 5-111.

Note

This directive does not generate any code. It is only an assertion by the programmer. The assembler produces error messages if any such assertions are inconsistent with each other, or with any vector notation in VFP data processing instructions.

The assembler faults vector notation in VFP data processing instructions following a `VFPASSERT SCALAR` directive, even if the vector length is 1.

Example

```
    VFPASSERT   SCALAR              ; scalar mode
    faddd       d4, d4, d0          ; okay
    fadds       s4<3>, s0, s8<3>    ; ERROR, vector in scalar mode
    fabss       s24<1>, s28<1>      ; ERROR, vector in scalar mode
                                    ; (even though length==1)
```

5.10.78 VFPASSERT VECTOR

The `VFPASSERT VECTOR` directive informs the assembler that following VFP instructions are in vector mode. It can also specify the length and stride of the vectors.

Syntax

`VFPASSERT VECTOR[<[`*`n`*`[:`*`s`*`]]>]`

where:

n	is the vector length, 1-8
s	is the vector stride, 1-2.

Usage

Use the `VFPASSERT VECTOR` directive to mark the start of a block of instructions where the VFP mode is `VECTOR`, and to mark changes in the length or stride of vectors.

Place the `VFPASSERT VECTOR` directive immediately after the instruction that makes the change. This is usually an `FMXR` instruction, but could be a `BL` instruction.

If a function expects the VFP to be in vector mode on entry, place a `VFPASSERT VECTOR` directive immediately before the first instruction. Such a function would not be ATPCS conformant. See the ATPCS chapter in *ADS Developer Guide* for further information.

See:
- *VFP directives and notation* on page 5-11
- *VFPASSERT SCALAR* on page 5-110.

Note

This directive does not generate any code. It is only an assertion by the programmer. The assembler produces error messages if any such assertions are inconsistent with each other, or with any vector notation in VFP data processing instructions.

Examples

```
    VFPASSERT VECTOR             ; vector mode, unspecified length
                                 ; and stride
    faddd  d4, d4, d0            ; ERROR, scalar in vector mode
    fadds  s16<3>, s0, s8<3>     ; okay
    fabss  s24<1>, s28<1>        ; okay (even though length==1)
```

```
VFPASSERT VECTOR<>          ; vector mode, unspecified length
                            ; and stride

VFPASSERT VECTOR<3>         ; vector mode, length 3, stride 1
faddd  d4, d4, d0           ; ERROR, scalar in vector mode
fadds  s24<3>, s0, s8<3>    ; okay
fabss  s24<1>, s24<1>       ; ERROR, wrong length

VFPASSERT VECTOR<4:2>       ; vector mode, length 4, stride 2
fadds  s8<4>, s0, s16<4>    ; ERROR, wrong stride
fabss  s16<4:2>, s28<4:2>   ; okay
fadds  s8<>, s2, s16<>      ; okay (s8 and s16 both have
                            ; length 4 and stride 2.
                            ; s2 is scalar.)
```

5.10.79 WEND directive

See *WHILE directive* on page 5-113.

5.10.80 WHILE directive

The `WHILE` directive starts a sequence of instructions or directives that are to be assembled repeatedly. The sequence is terminated with a `WEND` directive.

Syntax

`WHILE` *`logical-expression`*

`code`

`WEND`

where:

`logical-expression`

is an expression that can evaluate to either `{TRUE}` or `{FALSE}` (see *Logical expressions* on page 5-119).

Usage

Use the `WHILE` directive, together with the `WEND` directive, to assemble a sequence of instructions a number of times. The number of repetitions may be zero.

You can use `IF...ENDIF` conditions within `WHILE...WEND` loops.

`WHILE...WEND` loops can be nested (see *Nesting directives* on page 5-36).

Example

```
count   SETA    1                   ; you are not restricted to
        WHILE   count <= 4          ; such simple conditions
count   SETA    count+1             ; In this case,
            ; code                  ; this code will be
            ; code                  ; repeated four times
        WEND
```

5.11 Expressions, literals and operators

Expressions are combinations of symbols, values, unary and binary operators, and parentheses. There is a strict order of precedence in their evaluation:

1. Expressions in parentheses are evaluated first.
2. Operators are applied in precedence order.
3. Adjacent unary operators are evaluated from right to left.
4. Binary operators of equal precedence are evaluated from left to right.

The assembler includes an extensive set of operators for use in expressions. Many of the operators resemble their counterparts in high-level languages such as C (see *Unary operators* on page 5-120 and *Binary operators* on page 5-121).

This section contains the following subsections:
- *String expressions* on page 5-115
- *String literals* on page 5-115
- *Numeric expressions* on page 5-116
- *Numeric literals* on page 5-117
- *Floating-point literals* on page 5-118
- *Register-relative and program-relative expressions* on page 5-119
- *Logical expressions* on page 5-119
- *Logical literals* on page 5-119
- *Unary operators* on page 5-120
- *Binary operators* on page 5-121.

5.11.1 String expressions

String expressions consist of combinations of string literals, string variables, string manipulation operators, and parentheses. See:

- *String literals*
- *Variables* on page 5-31
- *SETS directive* on page 5-105
- *Unary operators* on page 5-120
- *String manipulation operators* on page 5-121.

Characters that cannot be placed in string literals can be placed in string expressions using the `:CHR:` unary operator. Any ASCII character from 0 to 255 is allowed.

The value of a string expression cannot exceed 512 characters in length. It may be of zero length.

Example

```
improb  SETS    "literal":CC:(strvar2:LEFT:4)
             ; sets the variable improb to the value "literal"
             ; with the left-most four characters of the
             ; contents of string variable strvar2 appended
```

5.11.2 String literals

String literals consist of a series of characters contained between double quote characters. The length of a string literal is restricted by the length of the input line (see *Format of source lines* on page 5-9).

To include a double quote character or a dollar character in a string, use two of the character.

C string escape sequences are also allowed, unless `-noesc` is specified (see *Command syntax* on page 5-4).

Examples

```
abc     SETS    "this string contains only one "" double quote"

def     SETS    "this string contains only one $$ dollar symbol"
```

5.11.3 Numeric expressions

Numeric expressions consist of combinations of numeric constants, numeric variables, ordinary numeric literals, binary operators, and parentheses. See:

- *Numeric constants* on page 5-31
- *Variables* on page 5-31
- *Numeric literals* on page 5-117
- *Binary operators* on page 5-121.

Numeric expressions can contain register-relative or program-relative expressions if the overall expression evaluates to a value that does not include a register or the program counter.

Numeric expressions evaluate to 32-bit integers. You may interpret them as unsigned numbers in the range 0 to $2^{32} - 1$, or signed numbers in the range -2^{31} to $2^{31} - 1$. However, the assembler makes no distinction between $-n$ and $2^{32} - n$. Relational operators such as >= use the unsigned interpretation. This means that 0 > –1 is `{FALSE}`.

Example

```
a   SETA    256*256             ; 256*256 is a numeric expression
    MOV     r1,#(a*22)          ; (a*22) is a numeric expression
```

5.11.4 Numeric literals

Numeric literals can take any of the following forms:

- *`decimal-digits`*
- `0x`*`hexadecimal-digits`*
- `&`*`hexadecimal-digits`*
- *`n_base-n-digits`*

where

`decimal-digits`
is a sequence of characters using only the digits 0 to 9.

`hexadecimal-digits`
is a sequence of characters using only the digits 0 to 9 and the letters A to F or a to f.

`n_` is a single digit between 2 and 9 inclusive, followed by an underscore character.

`base-n-digits`
is a sequence of characters using only the digits 0 to (n – 1).

You must not use any other characters. The sequence of characters must evaluate to an integer in the range 0 to $2^{32} – 1$ (except in `DCQ` and `DCQU` directives, where the range is 0 to $2^{64} – 1$).

Examples

```
a       SETA    34906

addr    DCD     0xA10E

        LDR     r4,&1000000F

        DCD     2_11001010

c3      SETA    8_74007

        DCQ     0x0123456789abcdef
```

5.11.5 Floating-point literals

Floating-point literals can take any of the following forms:

- `{-}`*`digits`*`E{-}`*`digits`*
- `{-}{`*`digits`*`}.`*`digits`*`{E{-}`*`digits`*`}`
- `0x`*`hexdigits`*
- `&`*`hexdigits`*

`digits`	are sequences of characters using only the digits 0 to 9. You can write `E` in uppercase or lowercase. These forms correspond to normal floating-point notation.
`hexdigits`	are sequences of characters using only the digits 0 to 9 and the letters A to F or a to f. These forms correspond to the internal representation of the numbers in the computer. Use these forms to enter infinities and NaNs, or if you want to be sure of the exact bit patterns you are using.

The range for single-precision floating point values is:

- maximum 3.40282347e+38
- minimum 1.17549435e–38.

The range for double-precision floating point values is:

- maximum 1.79769313486231571e+308
- minimum 2.22507385850720138e–308.

Examples

```
DCFD    1E308,-4E-100
DCFS    1.0
DCFD    3.725e15
LDFS    0x7FC00000              ; Quiet NaN
LDFD    &FFF0000000000000       ; Minus infinity
```

5.11.6 Register-relative and program-relative expressions

A register-relative expression evaluates to a named register plus or minus a numeric constant (see *MAP or ^ directive* on page 5-95).

A program-relative expression evaluates to the *program counter* (pc) plus or minus a numeric constant. It is normally a label combined with a numeric expression.

Example

```
        LDR     r4,=data+4*n    ; n is an assembly-time variable
        ; code
        MOV     pc,lr
data    DCD     value0
        ; n-1 DCD directives
        DCD     valuen          ; data+4*n points here
        ; more DCD directives
```

5.11.7 Logical expressions

Logical expressions consist of combinations of logical literals (`{TRUE}` or `{FALSE}`), logical variables, Boolean operators, relations, and parentheses (see *Boolean operators* on page 5-124).

Relations consist of combinations of variables, literals, constants, or expressions with appropriate relational operators (see *Relational operators* on page 5-123).

5.11.8 Logical literals

There are only two logical literals:

- `{TRUE}`
- `{FALSE}`.

5.11.9 Unary operators

Unary operators (Table 5-10) have the highest precedence (bind most tightly) and are evaluated first. A unary operator precedes its operand. Adjacent operators are evaluated from right to left.

Table 5-10 Operator precedence

Operator	Usage	Description
`?`	`?A`	Number of bytes of executable code generated by line defining symbol A.
`BASE`	`:BASE:A`	If A is a pc-relative or register-relative expression: **BASE** returns the number of its register component `BASE` is most useful in macros.
`INDEX`	`:INDEX:A`	If A is a register-relative expression: **INDEX** returns the offset from that base register. `INDEX` is most useful in macros.
`+` and `-`	`+A` `-A`	Unary plus. Unary minus. + and – can act on numeric and program-relative expressions.
`LEN`	`:LEN:A`	Length of string A.
`CHR`	`:CHR:A`	One-character string, ASCII code A.
`STR`	`:STR:A`	Hexadecimal string of A. `STR` returns an eight-digit hexadecimal string corresponding to a numeric expression, or the string `"T"` or `"F"` if used on a logical expression.
`NOT`	`:NOT:A`	Bitwise complement of A.
`LNOT`	`:LNOT:A`	Logical complement of A.
`DEF`	`:DEF:A`	`{TRUE}` if A is defined, otherwise `{FALSE}`.

-

5.11.10 Binary operators

Binary operators are written between the pair of subexpressions they operate on. Operators of equal precedence are evaluated in left to right order. The binary operators are presented below in groups of equal precedence, in decreasing precedence order.

Multiplicative operators

Multiplicative operators (Table 5-11) are the binary operators that bind most tightly and have the highest precedence:

Table 5-11 Multiplicative operators

Operator	Usage	Explanation
`*`	`A*B`	Multiply
`/`	`A/B`	Divide
`MOD`	`A:MOD:B`	A modulo B

These operators act only on numeric expressions.

String manipulation operators

String manipulation operators are shown in Table 5-12.

In the two slicing operators `LEFT` and `RIGHT`:

- `A` must be a string
- `B` must be a numeric expression.

In `CC`, `A` and `B` must both be strings.

Table 5-12 String manipulation operators

Operator	Usage	Explanation
`LEFT`	`A:LEFT:B`	The left-most B characters of A
`RIGHT`	`A:RIGHT:B`	The right-most B characters of A
`CC`	`A:CC:B`	B concatenated on to the end of A

Shift operators

Shift operators (Table 5-13) act on numeric expressions, shifting or rotating the first operand by the amount specified by the second.

Table 5-13 Shift operators

Operator	Usage	Explanation
`ROL`	`A:ROL:B`	Rotate A left by B bits
`ROR`	`A:ROR:B`	Rotate A right by B bits
`SHL`	`A:SHL:B`	Shift A left by B bits
`SHR`	`A:SHR:B`	Shift A right by B bits

Note

`SHR` is a logical shift and does not propagate the sign bit.

Addition, subtraction, and logical operators

Addition and subtraction operators act on numeric expressions.

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

Table 5-14 shows addition, subtraction, and logical operators.

Table 5-14 Addition, subtraction, and logical operators

Operator	Usage	Explanation
`+`	`A+B`	Add A to B
`-`	`A-B`	Subtract B from A
`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

Relational operators

Relational operators (Table 5-15) act on two operands of the same type to produce a logical value.

The operands may be:

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

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

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

Table 5-15 Relational operators

Operator	Usage	Explanation
`=`	`A=B`	A equal to B
`>`	`A>B`	A greater than B
`>=`	`A>=B`	A greater than or equal to B
`<`	`A<B`	A less than B
`<=`	`A<=B`	A less than or equal to B
`/=`	`A/=B`	A not equal to B
`<>`	`A<>B`	A not equal to B

Boolean operators

These are the weakest binding operators with the lowest precedence.

In all three cases both A and B must be expressions that evaluate to either `{TRUE}` or `{FALSE}`.

Table 5-16 Boolean operators

Operator	Usage	Explanation
`LAND`	`A:LAND:B`	Logical AND of A and B
`LOR`	`A:LOR:B`	Logical OR of A and B
`LEOR`	`A:LEOR:B`	Logical Exclusive OR of A and B

The Boolean operators perform the standard logical operations on their operands.

Chapter 6
The ARM Linker

This chapter describes the ARM linker, armlink. The full command syntax is given, in addition to reference information about armlink, memory maps, and scatter loading. This chapter contains the following sections:

6.1 About armlink

armlink, the ARM linker, enables you to:

- link a collection of objects and libraries into an executable image
- partially link a collection of objects into an object that can be used as input for a future link step
- specify where the code and data will be located in memory
- produce debug and reference information about the linked files.

Objects consist of input sections that contain code, initialized data, or the locations of memory that must be set to zero. Input sections can be *read-only* (RO), *read-write* (RW), or *zero-initialized* (ZI) These attributes are used by armlink to group input sections into bigger building blocks called output sections, regions and images. Output sections are approximately equivalent to ELF segments.

Image regions are placed in the system memory map at load time. Before you can execute the image, you might have to move some of its regions to their execution addresses. The memory map of an image therefore has two distinct views:

- the load view of the memory when the program and data are first loaded
- the execution view of the memory after code is moved to its final location.

See *Building blocks for objects and images* on page 6-16 for more information on the image hierarchy.

6.1.1 Input to armlink

Input to armlink consists of:

- One or more object files in ELF Object Format. This format is described in the the pdf documentation in the directory you use to install ADS (usually the root directory of the ADS CD).
- Optionally, one or more libraries created by armar as described in *ARM librarian* on page 7-9.

——— **Note** ———

For backward compatibility, armlink also accepts object files in AOF format and libraries in ALF format. These formats are obsolete and will not be supported in the future.

————————

6.1.2 Output from armlink

Output from a successful invocation of armlink is one of the following:

- an executable image in ELF executable format
- a partially linked object in ELF object format.

For simple images, ELF executable files contain segments that are approximately equivalent to RO and RW output sections in the image. An ELF executable file also has ELF sections that contain the image output sections.

An executable image in ELF executable format can be converted to other file formats by using the fromELF utility. See *The fromELF utility* on page 7-3 for more information.

Constructing an executable image

When you use armlink to construct an executable image, it:

- resolves symbolic references between the input object files
- extracts object modules from libraries to satisfy otherwise unsatisfied symbolic references
- sorts input sections according to their attributes and names, and merges similarly attributed and named sections into contiguous chunks
- eliminates duplicate copies of debug sections
- organizes object fragments into memory regions according to the grouping and placement information provided
- relocates relocatable values
- generates an executable image.

Constructing a partially linked object

When you use armlink to construct a partially linked object, it:

- eliminates duplicate copies of debug sections
- minimizes the size of the symbol table
- leaves unresolved references unresolved
- generates an object that can be used as an input to a subsequent link step.

6.1.3 Summary of armlink options

This section gives a brief overview of each armlink command-line option. The options are arranged into functional groups.

Accessing help and information

To get information on the available command-line options use:

```
-help
```

To get the tool version number use:

```
-vsn
```

Specifying the output type and the output file name

Use the following option to create a partially linked object instead of an executable image:

```
-partial
```

Name the output file using the following option:

```
-output
```

Specify the output file format using the following option:

```
-elf
```

Specifying memory map information for the image

Use the following options to specify simple memory maps:

```
-ro-base
-rw-base
-ropi
-rwpi
-split
```

For more complex images, use the option:

```
-scatter
```

Scatter loading is described in *Creating complex images with scatter loading* on page 6-39. See the *ADS Developer Guide* for examples of using `-scatter`, `-ro-base`, `-rw-base`, `-ropi`, `-rwpi` and `-split`.

The `-scatter` option is mutually exclusive with the use of any of the simple memory map options `-ro-base`, `-rw-base`, `-split`, `-ropi`, or `-rwpi`.

The memory map options cannot be used for partial linking because they specify the memory map of an executable.

Controlling image contents

These options control various miscellaneous factors affecting the image contents:

```
-debug | -nodebug
-entry
-first
-keep
-last
-libpath
-locals | -nolocals
-remove | -noremove
-scanlib | -noscanlib
```

Generating image-related information

These options control how you extract and present information about the image:

```
-info
-map
-symbols
-symdefs
-xref
-xreffrom
-xrefto
```

By default, armlink prints the information you requested on the standard output stream, `stdout`, but the information from all the commands can be redirected to a text file using the `-list` command-line option.

Controlling armlink diagnostics

These options control how armlink emits diagnostics:

```
-errors
-list
-verbose
-via
-strict
-unresolved
```

6.2 Armlink syntax

Note

For command-line arguments that use parentheses, you might need to escape the parentheses characters with a backslash (`\`) character on UNIX systems.

The complete linker command syntax is:

```
armlink [-help] [-vsn] [-partial] [-output file] [-elf]
[-ro-base address] [-ropi] [-rw-base address] [-rwpi] [-split]
[-scatter file] [-debug|-nodebug] [-remove (RO/RW/ZI)|-noremove]
[-entry location ] [-keep section-id] [-first section-id]
[-last section-id] [-libpath pathlist] [-scanlib|-noscanlib]
[-locals|-nolocals] [-info topics] [-map] [-symbols]
[-symdefs file] [-xref] [-xreffrom object(section)]
[-xrefto object(section)] [-errors file] [-list file] [-verbose]
[-via file] [-strict] [-unresolved symbol] [input-file-list]
```

where:

`-help`	This option prints a summary of some commonly used command-line options.
`-vsn`	This option displays the armlink version information.
`-partial`	This option creates a partially linked object instead of an executable image.
`-output` *file*	This option specifies the name of the output file. The file can be either a partially linked object or executable image. If the output file name is not specified, armlink uses the following defaults: `__image.axf` if the output is an executable image `__object.o` if the output is a partially-linked object. If *file* is specified without path information, it will created in the current working directory. If path information is specified, then that directory becomes the default output directory.
`-elf`	This option generates the image in ELF format. This is the only output format supported by armlink. This is the default.

`-ro-base` *`address`*

This option sets both the load and execution addresses of the region containing the RO output section at *`address`*. If this option is not specified, the default RO base address is 0x8000.

`-ropi`

This option makes the load and execution region containing the RO output section position independent. If this option is not used the region is marked as absolute. Usually each read-only input section must be read-only position independent. If this option is selected, armlink:

- checks that relocations between sections are valid
- ensures that any code generated by armlink itself, such as interworking veneers, is read-only position independent.

Note

The ARM tools cannot determine if the final output image will be Read-Only Position Independent until armlink finishes processing input sections. This means that armlink might emit ROPI error messages, even though you have selected the ROPI option for the compiler and assembler.

`-rw-base` *`address`*

This option sets the execution addresses of the region containing the RW output section at *`address`*.

If this option is used with `-split`, it sets both the load and the execution address of the region containing the RW output sections at *`address`*.

`-rwpi`

This option makes the load and execution region containing the RW and ZI output section position independent. If this option is not used the region is marked as absolute. The `-rwpi` option is ignored if `-rw-base` is not also used. Usually each writable input section must be read-write position independent. If this option is selected, armlink:

- checks that the PI attribute is set on input sections to any read-write execution regions
- checks that relocations between sections are valid
- generates sb-relative entries in Region$$Table and ZISection$$Table.

Note

The compiler does not force your writable data to be position-independent. This means that armlink might emit RWPI messages, even though you have selected the RWPI option for the compiler and assembler.

`-split`

This option splits the default load region, that contains the RO and RW output sections, into two load regions:

- one containing the RO output section. The default load address is 0x8000, but a different address can be specified with the `-ro-base` option.
- one containing the RW output section. The load address is specified with the `-rw-base` option. The `-split` option is ignored if `-rw-base` is not also used.

`-scatter` *`file`*

This option creates the image memory map using the scatter-loading description contained in *`file`*. The description provides grouping and placement details of the various regions and sections in the image. See *Creating complex images with scatter loading* on page 6-39.

`-debug`

This option includes debug information in the output file. The debug information includes debug input sections and the symbol and string table. This is the default.

`-nodebug`

This option turns off the inclusion of debug information in the output file. The image is smaller, but you cannot debug it at the source level. armlink discards any debug input section it finds in the input objects and library members, and does not include the symbol and string table in the image as loaded into the debugger. This only affects the image size as loaded into the debugger and has no effect on the size of any resulting binary image that is downloaded to the target.

If you are creating a partially linked object rather than an image, armlink discards the debug input sections it finds in the input objects, but does produce the symbol and string table in the partially linked object.

Note

fromELF cannot translate images produced without debug information into other file formats. It can only display the object or image contents as text.

Do not use `-nodebug` if you will be using fromELF to translate the ELF image into other formats.

`-remove (RO/RW/ZI)`

This option performs unused section elimination on the input sections to remove unused sections from the image. An input section is considered to be used if it contains the image entry point, or if it is referred to from a used section. See also *Unused section elimination* on page 6-29.

Caution

You must take care not to remove exception handlers when using `-remove`. Use the `-keep` option to identify exception handlers or label them as entry points.

You can use section attribute qualifiers for more precise control of the unused section elimination process. If a qualifier is used, it can be one or more of the following:

`RO`	remove all unused sections of type RO.
`RW`	remove all unused sections of type RW.
`ZI`	remove all unused sections of type ZI.

The qualifiers can appear in any case and order, but must be enclosed in parentheses `()`, and must be separated by a slash `/`.

The default is `-remove(RO/RW/ZI)`.

If no section attribute qualifiers are specified, all unused sections are eliminated. `-remove` is equivalent to `-remove (RO/RW/ZI)`.

`-noremove` This option does not perform unused section elimination on the input sections. This retains all input sections in the final image even if they are unused.

`-entry` *`location`* This option specifies the unique entry point of the image. The image can contain multiple entry points, but the entry point specified using this command is stored in the executable file header for use by the loader. There can be only one occurrence of this command on the command line. Replace *`location`* with one of the following:

`entry_address`

A numerical value, for example:

```
-entry 0x0
```

symbol This option specifies an image entry point as the address of *symbol*. If multiple definitions of *symbol* exist, armlink will generate an error. For example:

```
-entry int_handler
```

offset+object(*section*)

This option specifies an image entry point as an *offset* inside a *section* within a particular *object*. For example:

```
-entry 8+startup(startupseg)
```

There must be no spaces within the argument to `-entry`. The *input section* and object names are matched without case-sensitivity. You can use the following simplified notation:

- `object(section)` if `offset` is zero
- `object` if there is only one input section. If this form is used and there is more than one non-debug input section in `object`, armlink will generate an error.

`-keep` *section-id*

Specifies input sections that will not be removed by unused section elimination. See *Specifying an image memory map* on page 6-19. Replace *section-id* with one of the following:

symbol This option specifies that the input section defining *symbol* should be retained during unused section elimination. If multiple definitions of *symbol* exist, then all input sections that define *symbol* are treated similarly. For example:

```
-keep int_handler
```

object(*section*)

This option specifies that *section* from *object* should be retained during unused section elimination. The input section and object names are matched without case-sensitivity. For example:

```
-keep vectors.o(vect)
```

There can be multiple occurrences of this command on the command line.

`object` This option specifies that the single input section from `object` should be retained during unused section elimination. The object name is matched without case-sensitivity. If you use this short form and there is more than one input section in `object`, armlink will generate an error. For example:

```
-keep dspdata.o
```

There can be multiple occurrences of this command on the command line.

`-first section-id`

This option places the selected input section first its execution region. This can, for example, place the section containing the reset and interrupt vector addresses first in the image. Replace `section-id` with one of the following:

`symbol` Selects the section that defines `symbol`. You must not specify a symbol that has more than one definition, as more than one section cannot be placed first. For example:

```
-first reset
```

`object(section)`

Selects `section` from `object`. There must be no space between `object` and the following open parenthesis. For example:

```
-first init.o(init)
```

`object` Selects the single input section in `object`. If you use this short form and there is more than one input section, armlink will generate an error. For example:

```
-first init.o
```

When using scatter loading, use `+FIRST` in the scatter description file instead.

Using `-first` cannot override the basic attribute sorting order for output sections in regions that places RO first, RW second, and ZI last. If the region has an RO section, an RW or a ZI section cannot be placed first. If the region has an RO or RW section, a ZI section cannot be placed first.

Two different sections cannot both be placed first in the same execution region, so only one instance of this option is permitted.

`-last` *`section-id`*

This option places the selected input section last in its execution region. For example, this can force an input section that contains a checksum to be placed last in the RW section. Replace *`section-id`* with one of the following:

`symbol` Selects the section that defines *`symbol`*. You must not specify a symbol that has more than one definition, as more than one section cannot be placed last. For example:

```
-last checksum
```

`object``(`*`section`*`)`

Selects the *`section`* from *`object`*. There must be no space between *`object`* and the following open parenthesis. For example:

```
-last checksum.o(check)
```

`object` Selects the single input section from *`object`*. If there is more than one input section in `object`, armlink will generate an error.

When using scatter loading, use `+LAST` in the scatter description file instead.

Using `-last` cannot override the basic attribute sorting order for output sections in regions that places RO first, RW second, and ZI last. If the region has a ZI section, an RW section cannot be placed last. If the region has an RW or ZI section, an RO section cannot be placed last.

Two different sections cannot both be placed last in the same execution region, so only one instance of this option is permitted.

`-libpath` *`pathlist`*

This option specifies a list of paths that are used to search for libraries. These paths override the path specified by the `ARMLIB` environment variable. *`pathlist`* is a comma-separated list of paths *path1, path2,... pathn* that are used to search for required libraries. The default path for the directory containing the ARM libraries is specified by the `ARMLIB` environment variable. See *Library searching, selection and scanning* on page 6-26 for more information on including libraries.

`-scanlib` This option allows scanning of required libraries to resolve references. This is the default.

`-noscanlib` This option prevents the scanning of default libraries in a link step.

`-locals` This option adds local symbols to the output symbol table when producing an executable image. This is the default.

`-nolocals` This option does not add local symbols to the output symbol table when producing an executable image. This is a useful optimization if you want to reduce the size of the output symbol table.

`-info` *`topics`* This option prints information about specified topics, where *`topics`* is a comma-separated list of topic keywords. A topic keyword can be one of the following:

`sizes`	Gives a list of the Code and Data (RO Data, RW Data, ZI Data, and Debug Data) sizes for each input object and library member in the image. Using this option implies `-info sizes,totals`.
`totals`	Gives totals of the Code and Data (RO Data, RW Data, ZI Data, and Debug Data) sizes for input objects and libraries.
`veneers`	Gives details of armlink-generated veneers. For more information on veneers see *Veneer generation* on page 6-30.
`unused`	Lists all unused sections that were eliminated from the image as a result of using `-remove`.

Note

Spaces are not allowed between keywords in a list. For example, you can enter:

```
-info sizes,totals
```

but not:

```
-info sizes, totals
```

`-map` This option creates an image map. The image map contains addresses and sizes of each load region, execution region and input section in the image, including debugging and linker-generated input sections.

`-symdefs` *`file`*	This option creates a symbol definition file containing the global symbol definitions from the output image. This file can be used as input when linking another image. See *Accessing symbols in another image* on page 6-31 for more information. If *file* is specified without path information, it will searched for or created in the output directory, that is the directory where the output image is being written to.
`-symbols`	This option lists each local and global symbol used in the link step, and its value. This includes linker-generated symbols.
`-xref`	This option lists all cross-references between input sections.
`-xreffrom` *`object(section)`*	This option lists cross-references from input *`section`* in *`object`* to other input sections. This is a useful subset of the listing produced by using `-xref` if you are interested in references from a specific input section. You can have multiple occurrences of this option in order to list references from more than one input section.
`-xrefto` *`object(section)`*	This option lists cross-references to input *`section`* in *`object`* from other input sections. This is a useful subset of the listing produced by using `-xref` if you are interested in references to a specific input section. You can have multiple occurrences of this option in order to list references to more than one input section.
`-errors` *`file`*	Redirects the diagnostics from the standard error stream to *`file`*.
`-list` *`file`*	This option redirects the diagnostics from output of the `-info`, `-map`, `-symbols`, `-xref`, `-xreffrom`, and `-xrefto` commands to *`file`*. If *file* is specified without path information, it is created in the output directory, that is the directory the output image is being written to.
`-verbose`	This option prints messages indicating progress of the link operation.
`-via` *`file`*	This option reads a further list of input filenames and linker options from *`file`*. You can enter multiple `-via` options on the armlink command line. The `-via` options can also be included within a via file.

`-strict` This option strictly enforces memory attributes.

`-unresolved` *`symbol`*

This option matches each reference to an undefined symbol to the global definition of *`symbol`*. *`symbol`* must be both defined and global, otherwise it will appear in the list of undefined symbols, and the link step will fail. This option is particularly useful during top-down development, when it can be possible to test a partially-implemented system by matching each reference to a missing function to a dummy function.

This option does not display warnings.

`input-file-list` This is a space-separated list of objects or libraries.

A special type of object file, the symdef file, may be included in the list to provide global symbol values for a previously generated image file. See *Accessing symbols in another image* on page 6-31 for more information.

There are two ways you can use libraries in the input file list:

- Specify particular members to be extracted from a library and added to the image as individual objects. For example, specify `mystring.lib(strcmp.o)` in the input file list.
- Specify a library to be added to the list of libraries that is used to extract members if they resolve any non-weak unresolved references. For example, specify `mystring.lib` in the input file list. Other libraries are added to this list implicitly by armlink when it scans the default library directories and selects the closest matching library variants available. Members from the libraries in this list are added to the image only when they resolve an unresolved non-weak references. For more information see *Library searching, selection and scanning* on page 6-26.

armlink processes the file list in the following order:

1. Objects are added to the image unconditionally.
2. Members selected from libraries using patterns are added to the image unconditionally, as if they were objects.
3. Libraries are added to the list of libraries that is used to extract members if they resolve any non-weak unresolved references.

6.3 Image structure

The structure of an image is defined by:

- the number of its constituent regions and output sections
- the positions in memory of these constituent regions and sections when
  - — the image is loaded
  - — the image executes.

6.3.1 Building blocks for objects and images

The components of a partially-linked or executable file are constructed from a hierarchy of images, regions, output sections, and input sections.

- An image consists of one or more regions. Each region consists of one or more output sections.
- Each output section contains one or more input sections.
- Input sections are the code and data information in an object file.

Figure 6-1 shows the relationship between regions, output sections, and input sections.

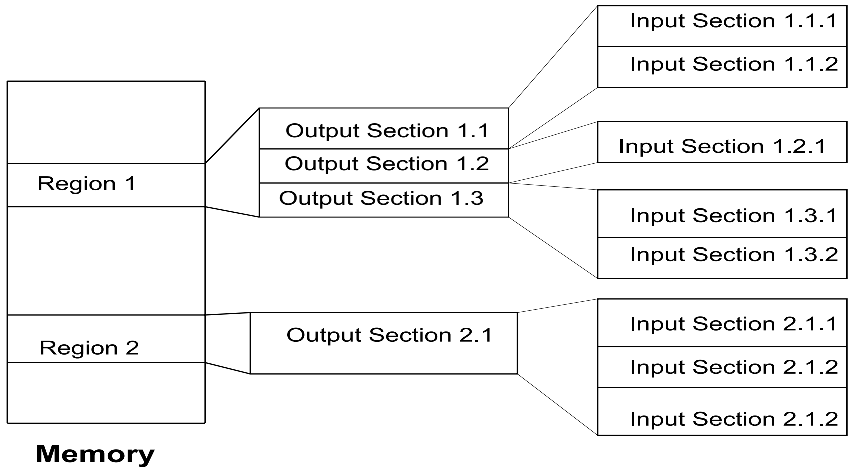


Figure 6-1 Building blocks for an image

Input Sections

An input section contains code or initialized data, or describes a fragment of memory that is not initialized or that must be set to zero before the image can execute. Input sections can have the attributes RO, RW, or ZI. These three attributes are used by armlink to group input sections into bigger building blocks called output sections and regions.

Output Sections

An output section is a contiguous sequence of input sections that have the same RO, RW, or ZI attribute. An output section has the same attributes as its constituent input sections. Within an output section, the input sections are sorted according to the rules described in *Ordering input sections by attribute* on page 6-21.

Regions A region is a contiguous sequence of one to three output sections. The output sections in a region are sorted according to their attributes. The RO output section is first, then the RW output section, and finally the ZI output section.

6.3.2 Load view and execution view of an image

Image regions are placed in the system memory map at load time. Before you can execute the image, you might have to move some of its regions to their execution addresses and create the ZI output sections. For example, initialized RW data might have to be copied from its load address in ROM to its execution address in RAM.

The memory map of an image has two distinct views as shown in Figure 6-2:

Load view This view describes each image region and section in terms of the address it is located at when the image is loaded into memory, that is the location before the image starts executing

Execution view This view describes each image region and section in terms of the address it is located at while the image is executing.

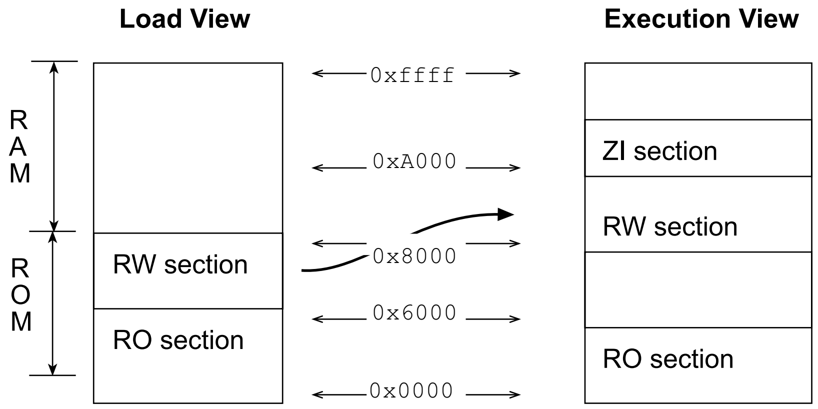


Figure 6-2 Load and execution memory maps

Table 6-1 compares the load and execution views.

Table 6-1 Comparing load and execution

Load	Description	Execution	Description
Load address	The address where a section, or region is loaded into memory before the image containing it starts executing. The load address of a section or a region can differ from its execution address.	Execution address	The address where a section or region is located while the image containing it is being executed.
Load region	A region in the load address space.	Execution region	A region in the execution address space.

6.3.3 Specifying an image memory map

An image can consist of any number of regions and output sections. Any number of these regions can have different load and execution addresses. To construct the memory map of an image, armlink must have information about:

- grouping, how input sections are grouped into output sections and regions
- placement, where image regions should be located in the memory maps.

Depending on the complexity of the memory maps of the image, there are two ways to pass this information to armlink:

Using command-line options

The following options can be used for simple cases where an image has only one or two load region and up to three execution regions:

- `-ro-base`
- `-rw-base`
- `-split`
- `-ropi`
- `-rwpi`.

The options listed above provide a simplified notation that gives the same settings as a scatter loading description for a simple image. For more information, see *Creating simple images* on page 6-34.

Using scatter loading description file

A description file is used for more complex cases where you require complete control over the grouping and placement of image components. This is described in full in *Creating complex images with scatter loading* on page 6-39. To use scatter loading, specify `-scatter` *`filename`* at the command line.

6.3.4 Image entry points

The entry point of an image is where the program execution can start spontaneously. The *initial* entry point for the image is a single value located in the header file.

For programs loaded into RAM by an operating system, the program loader starts the image execution by transferring control to the initial entry point in the image.

An embedded image can have multiple entry points, used by for example `Reset`, `IRQ`, `FIQ`, `SVC`, `UNDEF`, and `ABORT`, that can be used to transfer control when the image is running. If the embedded image is to be used by a loader however, it must have a single initial entry point specified in the header.

An image, that is the OS for a system for example, is loaded by the boot loader and entered at the initial entry point specified in the executable file header. After the image is loaded, the image overwrites the boot loader and becomes the OS. In this example there are many entry points, but only the initial execution point is specified in the executable file header.

You can specify any number of entry points in the image by marking the input section in the assembler sources with the `ENTRY` keyword.

armlink allows each object in an image to have one input section marked with the `ENTRY` keyword. Each of these input sections is treated as an entry point.

armlink accepts multiple occurrences of either of the following linker options:

`-entry offset +object(`*`input_section`*`)`

`-entry` *`symbol`*

Each of the locations identified is treated as an entry point.

In addition to specifying entry points in general, you can specify the initial entry point, that entry point will be placed in the executable file header. Use the `-entry entry_address` variant to specify the initial entry. There can be only one instance of this variant of the `-entry` option on the command line.

If you have not specified the initial entry point using an `-entry` *`entry_address`* option and the input objects contain only one entry point, armlink uses that entry point as the initial entry point for the image.

If you have not specified the initial entry point but more than one entry point has been specified, either by using the `-entry offset +object(`*`input_section`*`)` option or marking the input sections with `ENTRY`, none of the entry points are selected by armlink and the executable file header will not contain an initial entry point.

6.3.5 Section placement and sorting rules

armlink sorts all the input sections within a region according to their attributes. Input sections with identical attributes form a contiguous block within the region.

The base address of each section is determined by the sorting order defined by armlink.

While generating an image, armlink sorts the input sections in the following order:

- by attribute
- by input section name
- by their positions in the input list, except where overridden by a `-first` or `-last` option. This is described in *Using FIRST and LAST to place sections* on page 6-22.

By default, armlink creates an image consisting of an RO, an RW, and optionally a ZI, output section. The RO output section can be protected at runtime on systems that have memory management hardware.

Page alignment of the RO and RW output sections of the image can be forced using the section alignment attribute of areas. You set this using the `ALIGN` attribute of the ARM assembler `AREA` directive (see *Directives* on page 5-36).

Ordering input sections by attribute

Portions of the image associated with a particular language runtime system are collected together into a minimum number of contiguous regions. armlink orders input sections by attribute as follows:

- read-only code
- read-only data
- read-write code
- other initialized data
- zero-initialized (uninitialized) data.

Input sections that have the same attributes are ordered by their names. Names are considered to be case-sensitive and are compared in alphabetical order using the ASCII collation sequence for characters.

Identically attributed and named input sections are ordered according to their relative positions in the input list.

These rules mean that the positions of identically attributed and named input sections included from libraries are not predictable. If more precise positioning is required, you can extract modules manually, and include them in the input list.

Using FIRST and LAST to place sections

Within a region, all RO code input section sections are contiguous and form an RO output section that must precede the output section containing all the RW input sections.

If you are not using scatter loading, use the `-first` and `-last` linker options to place input sections.

If you are using scatter loading, use the pseudo-attributes `FIRST` and `LAST` in the scatter load description file to mark the first and last input sections in an execution region if the placement order is important.

However, `FIRST` and `LAST` must not violate the basic attribute sorting order. This means that an input section can be first (or last) in the execution region if the output section it is in is the first (or last) output section in the region. For example, in an execution region containing RO input sections, the `FIRST` input section must be an RO input section. Similarly, if the region contains any ZI input sections, the `LAST` input section must be a ZI input section.

Within each output section, input sections are sorted alphabetically according to their names, and then by their positions in the input order.

Aligning sections

When input sections have been ordered and the base address fixed, armlink can insert padding to force each input section to start at an address that is a multiple of:

$2^{\texttt{(input section alignment)}}$

Input sections are commonly aligned at word boundaries. Use the `ALIGN` directive in assembly language to control section alignment.

6.4 Linker-defined symbols

armlink defines some symbols that contain the character sequence `$$`. These symbols and all other external names containing the sequence `$$` are ARM-reserved names. These symbols are used to specify region base addresses, output section base addresses, and input section base addresses and their limits.

These symbolic addresses can be imported and used as relocatable addresses by your assembly language programs, or referred to as **`extern`** symbols from your C or C++ source code.

Note

Linker-defined symbols are defined by armlink only when your code references them.

6.4.1 Region-related symbols

Region-related symbols are generated when armlink is creating an image using a scatter loading description. The description names all the execution regions in the image, and provides their load and execution addresses (see *The scatter load description file* on page 6-40).

Table 6-2 shows the symbols that armlink generates for every execution region present in the image.

Table 6-2 Region-related linker symbols

Symbol	Description
`Load$$`*`region_name`*`$$Base`	Load address of the region.
`Image$$`*`region_name`*`$$Base`	Execution address of the region.
`Image$$`*`region_name`*`$$Length`	Execution region length in bytes (multiple of 4).

For every execution region containing a ZI output section, armlink generates two additional symbols, as shown in Table 6-3.

Table 6-3 Additional symbols for ZI sections

Symbol	Description
`Image$$`*`region_name`*`$$ZI$$Base`	Execution address of the ZI output section in this region.
`Image$$`*`region_name`*`$$ZI$$Length`	Length of the ZI output section in bytes (multiple of 4).

Note

The ZI output sections of an image are not created statically, but are automatically created dynamically. Therefore there is no load address symbol for ZI output sections.

6.4.2 Output section related symbols

The symbols shown in Table 6-4 are generated if command-line options are used to create a simple image. A simple image has three output sections (RO, RW and ZI) that produce the three executions regions.

Table 6-4 Section-related linker symbols

Symbol	Description
`Image$$RO$$Base`	Address of the start of the RO output section.
`Image$$RO$$Limit`	Address of the first byte beyond the end of the RO output section.
`Image$$RW$$Base`	Address of the start of the RW output section.
`Image$$RW$$Limit`	Address of the byte beyond the end of the ZI output section.
`Image$$ZI$$Base`	Address of the start of the ZI output section.
`Image$$ZI$$Limit`	Address of the byte beyond the end of the ZI output section.

——— Note ———

These symbols contain no useful information if a scatter load description is used to specify grouping and placement information. Code that uses these symbols while using a scatter-loaded image will not produce expected results. In such cases, only the region-related symbols described in *Region-related symbols* on page 6-23 should be used.

6.4.3 Input section related symbols

For every input section present in the image, armlink generates the symbols shown in Table 6-5.

Table 6-5 Area-related linker symbols

Symbol	Description
`SectionName$$Base`	Address of the start of the consolidated section called `SectionName`.
`SectionName$$Limit`	Address of the byte beyond the end of the consolidated section called `SectionName`.

——— Note ———

If your code refers to the input-section symbols, it is assumed that you expect all the input sections in the image with the same name to be placed contiguously in the image memory map. If your scatter loading description places these input sections non-contiguously, armlink will diagnose an error because the use of the base and limit symbols over non-contiguous memory will usually produce unpredictable and undesirable effects.

6.5 Library searching, selection and scanning

An object file can refer to external symbols that are, for example, functions or variables. armlink attempts to resolve these references by matching them to definitions found in other object files and libraries. armlink recognizes a collection of ELF files stored in an `ar` format file as a library.

The difference between the way armlink adds object files to the image and the way it adds libraries to the image:

- Each object file in the input list is added to the output image unconditionally, whether or not anything refers to it.
- A member from a library is included in the output only if an object file or an already-included library member makes a non-weak reference to it, or if armlink is explicitly instructed to add it.

Unresolved references to weak symbols do not cause library members to be loaded.

Note

If the `-noscanlib` option was specified, armlink does not search for libraries but uses only those libraries that are specified in the input file list to resolve references.

If a library member is explicitly requested in the input file list, it is loaded even if it does not resolve any current references. In this case, an explicitly requested member is treated as if it were an ordinary object.

armlink creates a list of libraries as follows:

1. armlink adds any libraries specified in the input file list to the list.
2. The user-specified search path is examined by armlink to identify directories containing the appropriate libraries. See *Searching for libraries* on page 6-27 for details on the search process.
3. The best-suited library variants are chosen from the searched directories and their subdirectories. ARM-supplied libraries have multiple variants that are named according to the attributes of their members. For details on the library variants see *Library naming conventions* on page 4-96 and *Selecting library variants* on page 6-27.

When armlink has constructed the list of libraries has been created, it repeatedly scans each library in the list to resolve references. See *Scanning the libraries* on page 6-28 for details.

6.5.1 Searching for libraries

For user libraries explicitly mentioned on the command line, a path is required if they are not in the current working directory. If a path is not given, the search paths `-libpath /ARMLIB` will *not* be used. You can specify the search paths by:

- Adding the `-libpath` argument to armlink command line with a comma-separated list of parent directories.

  This list must end with the parent directory of the ARM library directories `armlib` and `cpplib`. The `ARMLIB` variable holds the path to the ARM library parent directory.

  Caution

  `-libpath` overrides the paths specified by the ARMLIB variable.

- Using the CodeWarrior IDE linker configuration panel (see *CodeWarrior IDE Guide*).
- Using the environment variable `ARMLIB`.

armlink combines each parent directory, given by `-libpath`, the configuration panel, or the `ARMLIB` variable, with each subdirectory request from the input objects and identifies the place to search for the ARM library. The names of ARM subdirectories within the parent directories are placed in each compiled object by using a symbol of the form `Lib$$Request$$`*`sub_dir_name`*.

If a directory has been specified in a format ending with the directory separator, for example `c:\myapp\mylib\`, this path is used to search for user libraries.

If a directory has been specified in a format that does not end with the directory separator, for example `c:\ARM\Lib`, this path is used to search for the two subdirectories holding the ARM libraries: `armlib` and `cpplib`.

6.5.2 Selecting library variants

From each of the directories selected by armlink when searching for libraries, armlink must select the best-suited library. There are different variants of the ARM libraries based on the attributes of their member objects. The variant of the library is coded into the name of the library. See *Library naming conventions* on page 4-96.

armlink accumulates the attributes of each input object and uses them to select the library variant best suited to the accumulated attributes. If more than one of the selected libraries are equally suited, the library selected first is retained and the others are rejected.

The final list contains all the libraries that armlink will scan in order to resolve references.

6.5.3 Scanning the libraries

When all the directories have been searched, and the most compatible library variants have been selected and added to the list of libraries, each of the libraries is scanned to load the required members:

1. armlink searches for each currently unsatisfied non-weak reference sequentially through the list of libraries.

   The sequential nature of the search ensures that armlink chooses the library that appears earlier in the list if two or more libraries define the same symbol. this enables you to override function definitions from other libraries, for example the ARM C libraries, by adding your libraries in the input file list.

2. If a library contains members that resolve any references, the members are loaded. As each such member is loaded it might satisfy some unresolved references, possibly including weak ones.

3. Loading a library member might also create new unresolved weak or non-weak references.

4. The process continues until the non-weak references are either resolved or are incapable of being resolved by any library.

If any non-weak reference remains unsatisfied at the end of the scanning operation, armlink generates an error message.

6.6 Optimizations and modifications

armlink performs some optimizations and modifications in order to remove duplicate sections and allow interworking between ARM and Thumb code.

6.6.1 Common debug section elimination

The compilers and assemblers generate one set of debug sections per source file. armlink can detect multiple copies of the set of debug sections and discard all but one copy in the final image. This can result in a considerable reduction in image debug size.

6.6.2 Common section elimination

If there are inline functions or templates used in the source, the ARM C++ compilers generate complete objects for linking such that each object contains the out-of-line copies of inline functions and template functions that the object requires. When these functions are declared in a common header file, the functions might have be defined many times in separate objects that are subsequently linked together. In order to eliminate duplicates, the compilers compile these functions into separate instances of common code sections and armlink retains just one copy of each common code section.

It is possible that the separate instances of a common code section are not identical. Some of the copies, for example, may be found in a library which has been built with different (but compatible) build options, different optimization, or different debug options.

If the copies are not identical, armlink retains the best available variant of each common code section based on the attributes of the input objects.

6.6.3 Unused section elimination

Unused section elimination removes code that is never executed, or data that is not referred to by the code, from the final image. This optimization can be controlled by the `-remove` and `-noremove` options.

An input section is retained in the final image in the following conditions:

- if it contains an entry point
- if it is referred to, directly or indirectly, by a non-weak reference from an input section containing an entry point
- if it was specified as the first or last input section by the `-first` or `-last` option
- if it has been marked as unremovable by the `-keep` option.

6.6.4 Veneer generation

armlink must generate veneers when:

- a branch involves change of state between ARM state and Thumb state
- a branch involves a destination beyond the branching range of the current state.

A veneer can extend the range of branch and change state. armlink combines long branch capability into the state change capability. All interworking veneers are also long branch veneers.

There are four types of veneers to handle different branching requirements:

ARM to ARM	Long branch capability
ARM to Thumb	Long branch capability and interworking capability
Thumb to ARM	Long branch capability and interworking capability
Thumb to Thumb	Long branch capability.

armlink creates one input section called `Veneer$$Code` for each veneer. A veneer is generated only if no other existing veneer is able to satisfy the requirements. If two input sections contain a long branch to the same destination, only one veneer will be generated if the veneer can be reached by both sections.

All veneers cannot be collected into one input section because the resulting veneer input section might not to be within range of other input sections. If the sections are not within addressing range, long branching is not possible.

6.7 Accessing symbols in another image

If you want one image to know the global symbol values of another image, you can use a symdefs file.

This can be used, for example, if you have one image that always resides in ROM and multiple images that are loaded into RAM. The images loaded into RAM can access global functions and data from the image located in ROM.

6.7.1 Reading a symdefs file

A symdefs file can be considered as an object file with symbol information but no code or data. To read a symdefs file, add it to your file list as you would any object file. armlink reads the file and adds the symbols and their values to the output symbol table. The added symbols have the `ABSOLUTE GLOBAL` attributes.

If a partial link is being performed, the symbols will be added to the output object symbol table. If a full link is being performed, the symbols will be added to the image symbol table.

armlink generates errors for invalid rows in the file. A row is invalid if:

- any of the columns are missing
- any of the columns have invalid values.

The symbols extracted out of a symdefs file are treated in exactly the same way as symbols extracted from an object symbol table. The same restriction regarding multiple symbol definitions and ARM/Thumb synonyms apply.

6.7.2 Creating a symdefs file

armlink produces a symdefs file during a successful final link stage. It is not produced for partial linking or for unsuccessful final linking.

Use armlink option `-symdefs` *`filename`* to produce a symdefs file. If *filename* does not exist, the file will be created containing all global symbols. If *filename* exists, the existing contents of *filename* will be used.

Outputting a subset of the global symbols

By default, all global symbols are written to the symdefs file.

When *filename* exists, armlink uses its contents to restrict the output to a subset of the global symbols. To restrict the output symbols:

1. Specify `-symdefs` *`filename`* when you are doing a final link for *image1*. armlink creates a symdef file *filename*.
2. Open *filename* and remove any symbol entries you do not want in the final list.
3. Specify `-symdefs` *`filename`* when you are doing a final link for *image2*.

   You can also edit *filename* and then link *image1* again to, for example, update the symbol list after one or more objects used to create *image1* have changed.
4. armlink creates a temporary output file.
5. The comments and blank lines from *filename* are copied to the temporary file.
6. When a symbol is found in *filename*, the symbol is output to the temporary file using the address of the symbol in the current image.
7. If a symbol is in *filename* more than once, only one occurrence of the symbol is placed in the temporary file.
8. If a symbol is found in *filename* but does not exist in the current image, no output will be produced for that symbol.
9. If the final link is successful, *filename* will be deleted and the temporary file renamed as *filename*.

6.7.3 Symdefs file format

The symdefs file is a type of object file that contains symbols and their values. Unlike other object files, however, it does not contain any code or data.

The file consists of an identification line, optional comments, and symbol information as shown in Example 6-1.

Example 6-1

```
#SYMDEFS# ARM Linker, ADS1.0 [Build 105] Last Updated Tues Aug 03 13:48:47 1999
;value type name, this is an added comment
0x00001000 A function1
0x00002000 T function2
   # This is also a comment, blank lines are ignored

0x00003300 A function3
0x00003340 D table1
```

Identifying string

If the first 11 characters in the text file are `#SYMDEFS#`, armlink recognizes the file as a symdefs file.

The identifying string is followed by linker version information and date and time of last update of the symdefs file. The version and update information are not part of the identification string.

Comments

You can insert comments manually with a text editor. Comments have the following properties:

- Any line where the first non-whitespace character is `;` or `#` is a comment.
- The first line must start with the special identifying comment `#SYMDEFS#`. This comment is inserted by armlink when the file is produced and must not be manually deleted.
- A `;` or `#` after the first non-whitespace character does not start a comment.
- Blank lines are ignored and can be inserted to improve readability.

Symbol information

The symbol information is provided by the address, type and name of the symbol on a single line.

Symbol value armlink writes the absolute address of the symbol in fixed hexadecimal format, for example `0x00004000`. If you edit the file, you can use either hexadecimal or decimal formats for the address value.

Type flag The single letter for type is:

A	ARM code
T	Thumb code
D	Data.

Symbol name

The name of the symbol.

6.8 Creating simple images

A simple image consists of a number of input sections of type RO, RW, and ZI. These input sections are collated to form the RO, the RW, and the ZI output sections. Depending on how the output sections are arranged within load and execution regions, there are three basic types of simple images

Type 1	One region in load view, three contiguous regions in execution view.
Type 2	One region in load view, three non-contiguous regions in execution view.
Type 3	Two regions in load view, three non-contiguous regions in execution view.

In all three simple image types, there are up to three execution regions. The first execution region contains the RO output section, the second execution region contains the RW output section (if present), and the third execution region contains the ZI output section (if present). These execution regions are referred to as the RO, the RW, and the ZI execution region.

6.8.1 Type 1: one load region and contiguous output regions

An image of this type consists of a single load region in the load view and the three execution regions are placed contiguously in the memory map. This approach is suitable for systems that load programs into RAM, Angel, for example OS bootloader, or desktop system (see Figure 6-3 on page 6-35).

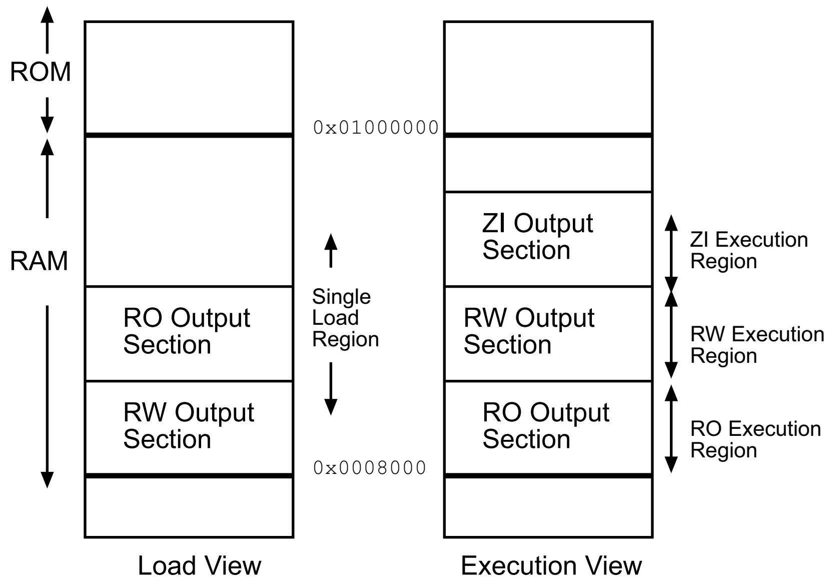


Figure 6-3 Type 1

Load view

The single load region consists of the RO and RW output sections placed consecutively. The ZI output section does not exist at load time. It is created before execution using the output section description in the image file.

Execution view

The three execution regions containing the RO, RW, and ZI output sections arranged contiguously. The execution address of the RO and RW execution regions are the same as their load addresses, so nothing has to be moved from its load address to its execution address. However, the ZI execution region that contains the ZI output section is created before execution begins.

Use armlink option `-ro-base` *`address`* to specify the load and execution address of the region containing the RO output.

6.8.2 Type 2: one load region and non-contiguous output regions

An image of this type consists of a single load region, and three execution regions in execution view. The RW execution region is not contiguous with the RO execution region. This approach is used, for example, for simple ROM-based embedded systems (see Figure 6-4).

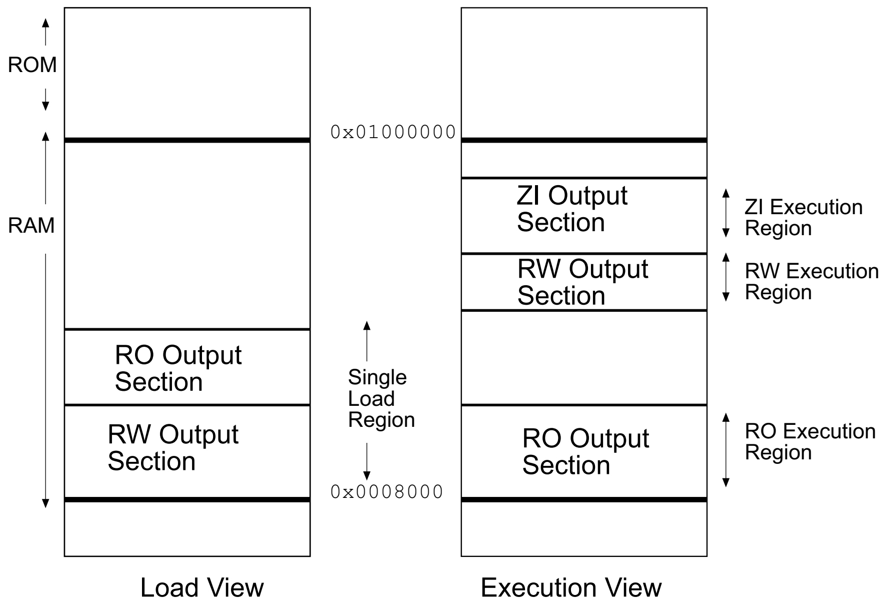


Figure 6-4 Type 2

Load view

In the load view, the single load region consists of the RO and RW output sections placed consecutively, in ROM for example. The ZI output section does not exist at load time. It is created before execution using the description of the output section contained in the image file.

Execution view

In the execution view, the first execution region contains the RO output section and the second execution region contains the RW and ZI output sections.

The execution address of the region containing the RO output section is the same as its load address, so the RO output section does not have to be moved.

The execution address of the region containing the RW output section is different from its load address, so the RW output section is moved from its load address (from the single load region) to its execution address (into the second execution region). The ZI execution region, and its output section, is placed contiguously with the RW execution region.

Use armlink options `-ro-base` *`address`* to specify the load and execution address for the RO output section and `-rw-base` *`exec_address`* to specify the execution address of the RW output section. If `-ro-base` option is not used to specify the address, the default value of 0x8000 is used by armlink.

6.8.3 Type 3: two load regions and non-contiguous output regions

This type of image is similar to images of type 2 except that the single load region in type 2 is now split into two load regions (see Figure 6-5).

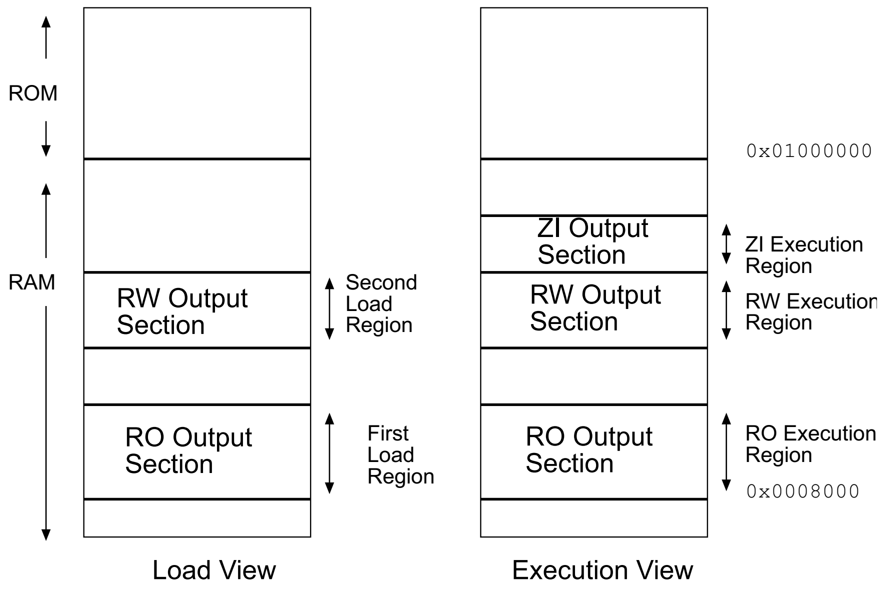


Figure 6-5 Type 3

Load view

In the load view, the first load region consists of the RO output section, and the second load region consists of the RW output section. The ZI output section does not exist at load time. It is created before execution using the description of the output section contained in the image file.

Execution view

In the execution view, the first execution region contains the RO output section and the second execution region contains the RW and ZI output sections.

The execution address of the RO region is the same as its load address, so the contents of the RO output section are not moved from their load address to their execution address.

The execution address of the RW region is also the same as its load address, so the contents of the RW output section are not moved from their load address to their execution address. However, the ZI output section is created before execution begins and placed after the RW region.

Specify the load and execution address using the following linker options:

`-split` This option splits the default single load region (that contains the RO and RW output sections) into two load regions, one containing the RO output section and one containing the RW output section.

`-ro-base` *`address`*

This option instructs armlink to set the load and execution address of the region containing the RO section at *`address`* (for example, the address of the first location in ROM). If `-ro-base` option is not used to specify the address, the default value of 0x8000 is used by armlink.

`-rw-base` *`address`*

This option instructs armlink to set the execution address of the region containing the RW output section at *`address`*. If this option is used with `-split`, both the load and execution addresses of the RW region are specified.

-

6.9 Creating complex images with scatter loading

An image is made up of regions and output sections. Every region in the image can have a different load and execution address (see *Image structure* on page 6-16).

The scatter loading mechanism enables you to specify the memory map of an image to armlink. Scatter loading gives you complete control over the grouping and placement of image components. It is capable of describing complex image maps consisting of multiple regions scattered in the memory map at load and execution time. Figure 6-6 shows an example of a complex memory map.

To construct the memory map of an image, armlink must have:

- grouping information describing how input sections are grouped into regions
- placement information describing the addresses where image regions should be located in the memory maps.

You specify this information using a scatter load description in a text file that is passed to armlink.

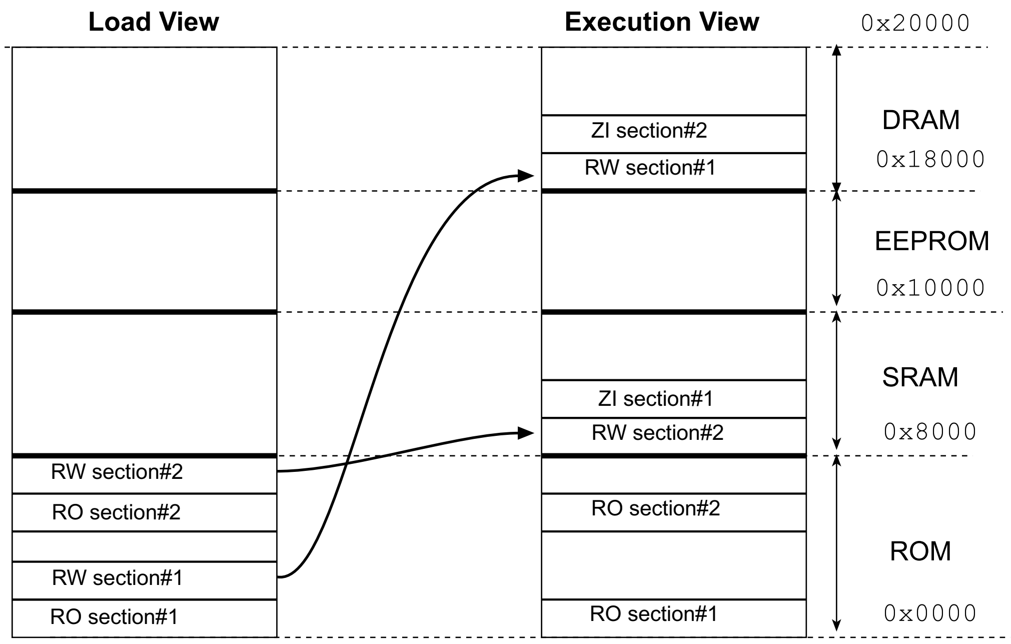


Figure 6-6 Scatter loaded memory map

6.9.1 Symbols defined for scatter loading

When armlink is creating an image using a scatter load description, it creates some region-related symbols. These are described in *Region-related symbols* on page 6-23. These special symbols are created by armlink only if your code references them.

6.9.2 Command-line option

armlink command-line option for using scatter loading is:

`-scatter` *`description_file_name`*

This instructs armlink to construct the image memory map as described in *`description_file_name`*. The format of the description file is given in *The scatter load description file* on page 6-40.

6.9.3 The scatter load description file

The scatter load description is a text file that describes to armlink the memory map of the image. The description file enables you to specify:

- the load address and maximum size of each load region
- the attributes of each load region
- the execution regions derived from each load region
- the execution address and maximum size of each execution region
- the memory access attributes of each execution region
- the attributes of each execution region
- the input sections for each execution region.

The description file format reflects the hierarchy of load regions, execution regions, and input sections.

Note

The assignment of input sections to regions is completely independent of the order patterns are written in the scatter load description.

The description itself is a sequence of tokens, whitespace, and comments, as shown in Table 6-7 on page 6-42. In the BNF punctuation in Table 6-6 has the usual significance.

Table 6-6 BNF syntax

Symbol	Description
A ::= *B*	Defines *A* as *B*
[*A*]	Optional element *A*
A+	Element *A* can have one or more occurrences
*A**	Element *A* can have zero or more occurrences
A | *B*	Either element *A* or *B* can occur
(*A B*)	Element *A* and *B* are grouped together

A scatter-load description is a sequence of load-region descriptions as defined below (in BNF):

```
Scatter-description ::= load-region-description+

load-region-description ::= load-region-name
base-designator [attribute-list] [ max-size ]
LBRACE execution-region-description+ RBRACE

execution-region-description ::= exec-region-name
base-designator [attribute-list] [max-size]
LBRACE input-section-description* RBRACE

base-designator ::= base-address  | (PLUS offset)

input-section-description ::=
module-selector-pattern [ LPAREN input-selectors RPAREN ]
```

```
input-selectors ::=
(PLUS input-section-attrs|input-section-pat )
([COMMA] PLUS input-section-attrs|COMMAinput-section-pat)*
```

Table 6-7 Scatter load description

Item	Description	
Special character	Single characters with special significance are:	
	`(`	`LPAREN`
	`)`	`RPAREN`
	`{`	`LBRACE`
	`}`	`RBRACE`
	`"`	`QUOTE`
	`,`	`COMMA`
	`+`	`PLUS`
	`;`	`SEMIC`
Tokens	Tokens are:	`LPAREN` `RPAREN` `LBRACE` `RBRACE` `QUOTE` `COMMA` `PLUS` `SEMIC`
Comments	Comments begin with a `SEMIC` and extend to the end of the current line. This means that a `WORD` cannot begin with a `SEMIC` (unless the `WORD` and `SEMIC` are enclosed in `QUOTE`s).	
Numbers	A `NUMBER` encodes a 32-bit unsigned value, and has one of the forms:	
	Prefix:	Number:
	O	octal-digit+
	&	hex-digit+
	ox	hex-digit+
	Ox	hex-digit+
		decimal-digit+
Word	A `WORD` is an alternation of quoted and unquoted `WORD`-segments:	
	Unquoted `WORD` segment	terminates on the first character in the set {Whitespace, `LPAREN`, `RPAREN`, `LBRACE`, `RBRACE`, `COMMA`, `PLUS`, `QUOTE`}.
	Quoted `WORD` segment	is enclosed by `QUOTE` characters and can contain any characters except *`newline`*. All other characters where `isspace()` is true are translated to `space`. Two consecutive `QUOTE`s stand for the literal `QUOTE` character and do not begin or end a quoted `WORD`-segment.

-

Load region description

A load region has:

- a name
- a base address
- attributes (optional)
- a maximum size (optional)
- a list of execution regions.

The syntax, in BNF, is:

```
load-region-description ::= load-region-name
base-designator [attribute-list] [ max-size ]
        LBRACE execution-region-description+ RBRACE

base-designator ::= base-address  | (PLUS offset)
```

where:

load-region-name

This names the load region. Only the first 31 characters are significant.The name is only used to identify each region. Unlike the *exec-region-name*, the *load-region-name* is not used to generate Load$$*region-name* symbols.

Note

An image created for use a debugger requires a unique base address for each region because the debugger must load regions at their load addresses. Overlapping (or zero) load region addresses result in part of the image being overwritten.

A loader or operating system, however, can correctly load PI regions.

base-designator This describes the base address:

base-address

Is the address where objects in the region should be linked. *base-address* must be a word-aligned NUMBER.

+offset Describes a base address that is *offset* bytes beyond the end of the preceding load region. The length of a region is always a multiple of four bytes, so *offset* must also be a multiple of four bytes. If this is the first in the load region, then *+offset* means that the base address begins at a point *offset* bytes after zero.

attribute-list

This specifies the properties of the load region contents:

PI	Position independent
RELOC	Relocatable
OVERLAY	Overlaid
ABSOLUTE	Absolute address

Only one of these attributes can be specified. The default load region attribute is ABSOLUTE.

load regions that have either of PI, RELOC or OVERLAY attributes are allowed to have overlapping address ranges. armlink faults overlapping address ranges for ABSOLUTE load regions.

The OVERLAY keyword allows you to have multiple exec regions at the same address. ARM does not support an overlay mechanism. In order to use multiple exec regions at the same address, you must have your own overlay manager.

max-size This specifies the maximum size of the load region. (If the optional max-size value is specified, armlink generates an error if the region has more than max-size bytes allocated to it.) The default value of max-size is 0xffffffff.

execution-region-description

This is a name and a base address, see *Execution region description* on page 6-45.

Execution region description

An execution region is described by a name and a base address.

The syntax, in BNF, is:

```
execution-region-description ::= exec-region-name
base-designator [attribute-list] [max-size]
        LBRACE input-section-description* RBRACE

base-designator ::= base-address  | (PLUS offset)
```

where:

exec-region-name

This names the execution region. (Only the first 31 characters are significant.)

base-designator This describes the base address:

base-address

Is the address where objects in the region should be linked. *base-address* must be a word-aligned `NUMBER`.

+*offset* Describes a base address that is *offset* bytes beyond the end of the preceding execution region. The length of a region is always a multiple of four bytes, so *offset* must also be a multiple of four bytes. If there is no preceding execution region (that is, if this is the first in the load region) then +*offset* means that the base address begins at a point *offset* bytes after the base of the containing load region.

attribute-list This specifies the properties of the execution region contents:

`PI`	Position independent
`RELOC`	Relocatable
`OVERLAY`	Overlaid
`ABSOLUTE`	Absolute address
`UNINIT`	Uninitialized data

Only one of the attributes `PI`, `RELOC`, `OVERLAY`, and `ABSOLUTE` can be specified. Unless one of the attributes `PI`, `RELOC`, or `OVERLAY` is specified, `ABSOLUTE` is an attribute of the execution region.

Execution regions that use the +*offset* form of *base-designator* either inherit the attributes of the preceding execution region, (or of the containing load region if this is the first execution region in the load region), or have the `ABSOLUTE` attribute. It is not possible for an execution region that uses the +*offset* form of *base-designator* to have its own attributes (except that the `ABSOLUTE` attribute prevents such inheritance).

Execution regions that have one of `PI`, `RELOC` or `OVERLAY` attributes are allowed to have overlapping address ranges. armlink faults overlapping address ranges for `ABSOLUTE` execution regions.

`UNINIT` specifies that the ZI output section, if any, in the execution region will not be initialized to zero. Use this to create execution regions containing unitialized data or memory-mapped I/O.

`max-size` This is an optional number that instructs armlink to generate an error if the region has more than `max-size` bytes allocated to it.

`input-section-description`

This is described in *Input section description*.

Input section description

An `input-section` description is a pattern that identifies input sections by:

- Module name (object file name, library member name, or library file name). The module name can use wildcard characters.
- Input section name, or input section attributes such as `READ-ONLY`, or `CODE`.

Note

- Only input sections that match both the `module-selector-pattern` and at least one *`section-selector`* are included in the execution region.

  If you omit `LPAREN` *`section-selectors`* `RPAREN`, the default is `+RO`.

- Do not rely on input section names generated by the compilers, or used by ARM library code. These can change between compilations if, for example, different compiler options are used. In addition, section naming conventions used by the compilers are not guaranteed to remain constant between releases.

The syntax is:

```
input-section-description ::=
module-selector-pattern [ LPAREN input-selectors RPAREN ]
```

where:

module-selector-pattern

This is a pattern constructed from literal text. The wildcard character `*` matches zero or more characters and `?` matches any single character. To match all objects for example, use:

```
*.o
```

An input section matches a *module-selector-pattern* when the *module-selector-pattern* matches one of the following:

- the name of the object file containing the section
- the name of the library member, without leading pathname
- the full name of the library the section was extracted from.

Matching is case-insensitive, even on hosts with case-sensitive file naming.

input-selectors

This is a comma-separated list of expressions. The syntax is described in *Input selectors* on page 6-47.

Input selectors

The input selector is a comma-separated list of patterns that input section names or attributes are matched against.

If you are specifying the pattern that will match the input section name, the name must be preceded by a `PLUS` (+). You can omit any comma immediately followed by a `PLUS`.

The syntax is:

```
input-selectors ::=
(PLUS input-section-attrs|input-section-pat )
([COMMA] PLUS input-section-attrs|COMMAinput-section-pat)*
```

where:

input-section-attrs

This is an attribute selector matched against the input section attributes. Each *input-section-attrs* follows a PLUS.

The selectors are not case-sensitive. The following selectors are recognized:

- RO-CODE
- RO-DATA
- RO, selects both RO-CODE and RO-DATA
- RW-DATA
- RW, selects both RW-CODE and RW-DATA
- ZI
- ENTRY, that is a section containing an ENTRY point.

The following synonyms are recognized:

- CODE for RO-CODE
- CONST for RO-DATA
- TEXT for RO
- DATA for RW
- BSS for ZI.

The following pseudo-attributes are recognized:

- FIRST
- LAST.

FIRST and LAST can be used to mark the first and last sections in an execution region if the placement order is important (for example, if a specific input section must be first in the region and an input section containing a checksum must be last). The first occurrence of FIRST or LAST as a *section-attrs* terminates the list.

The special module-selector pattern .ANY allows you to assign input sections to execution regions irrespective of their parent module, in order to fill up the execution regions with *don't care* assignments.

The *input-section-descriptions* having the .ANY module-selector pattern are resolved after all other (non-.ANY) input-section descriptions have been resolved and input sections have been assigned to the closest matching execution region.

Each remaining unassigned input section is assigned to the execution region with the following characteristics:

- the biggest remaining space (determined by the *max-size* and the sizes of the input sections already assigned to it)
- a matching `.ANY` *input-section-description*
- memory access attributes (if they exist) matching the memory attributes of the input section.

`input-section-pat`

This is a pattern that is matched, without case sensitivity, against the input section name. It is constructed from literal text, the wildcard characters `*` matches 0 or more characters, and `?` matches any single character.

6.9.4 Selecting veneer input sections in scatter loading descriptions

You can provide placement information for the veneer input sections. At most, one execution region in the scatter loading description can have the `*(Veneer$$Code)` section selector.

armlink places any veneer input section that can be safely put into the specified region. It might not be possible for a veneer input section to be assigned to the region because of address range problems or execution region size limitations. If the veneer cannot be added to the specified region, it is added to the execution region containing the relocated input section that generated the veneer.

Instances of `*(IWV$$Code)` in old style scatter loading descriptions are automatically translated into `*(Veneer$$Code)`. Use `*(Veneer$$Code)` in new descriptions.

6.9.5 Resolving multiple matches

If a section matches more than one execution region, the matches are resolved as described below. If a unique match cannot be found, armlink faults the scatter description. Each section is selected by a *`module-selector-pattern`* and an *`input-section-selector`*.

The following terminology is used to describe multiple matches:

- *`m1`* and *`m2`* represent module-selector-patterns
- *`s1`* and *`s2`* represent input-section-selectors.

In the case of multiple matches, armlink determines the region to assign the input section to on the basis of the *`module-selector-pattern`* and *`input-section-selector`* pair that is the most specific.

-

For example, if input section A matches *m1,s1* for execution region R1, and A matches *m2,s2* for execution region R2, armlink:

- assigns A to R1 if *m1,s1* is more specific than *m2,s2*
- assigns A to R2 if *m2,s2* is more specific than *m1,s1*
- diagnoses the scatter description as faulty if *m1,s1* is not more specific than *m2,s2* and *m2,s2* is not more specific than *m1,s1*.

The sequence armlink uses to determine the most specific `module-selector-pattern`, `input-section-selector` pair is as follows:

1. For the module selector patterns:

   m1 is more specific than *m2* if the text string *m1* matches pattern *m2* and the text string *m2* does not match pattern *m1*.

2. For the input section selectors:
   - If *s1* and *s2* are both patterns matching section names, the same definition as for module selector patterns is used.
   - If one of *s1*, *s2* matches the input section name and the other matches the input section attributes, *s1* and *s2* are unordered and the description is diagnosed as faulty.
   - If both *s1* and *s2* match input section attributes, `s1` is more specific than *s2* is defined by the relationships below:

     `ENTRY` is more specific than `RO-CODE`
     `ENTRY` is more specific than `RO-DATA`
     `ENTRY` is more specific than `RW-CODE`
     `ENTRY` is more specific than `RW-DATA`
     `RO-CODE` is more specific than `RO`
     `RO-DATA` is more specific than `RO`
     `RW-CODE` is more specific than `RW`
     `RW-DATA` is more specific than `RW`

     There are no other members of the (*s1* more specific than *s2*) relationship between section attributes.

3. For the `module-selector-pattern,input-section-selector` pair, *m1,s1* is more specific than *m2,s2* only if any of the following are true:
   - *s1* is a literal input section name (that is, it contains no pattern characters) and *s2* matches input section attributes other than `+ENTRY`
   - *m1* is more specific than *m2*
   - *s1* is more specific than *s2*

This matching strategy has the following consequences:

- Descriptions do not depend on the order they are written in the file.
- Generally, the more specific the description of an object, the more specific the description of the input sections it contains.
- The `input-section-selectors` are not examined unless:
  - — Object selection is inconclusive.
  - — One selector fully names an input section and the other selects by attribute. In this case, the explicit input section name is more specific than any attribute, other than `ENTRY`, that selects exactly one input section from one object. This is true even if the object selector associated with the input section name is less specific than that of the attribute.

6.9.6 Scatter loading descriptions for simple images

The command-line options (`-ro-base`, `-rw-base`, `-split`, `-ropi`, and `-rwpi`) create the simple image types described in *Specifying an image memory map* on page 6-19.

You can create the same image types by using the `-scatter` command-line option and a file containing one of four corresponding scatter load descriptions .

Type 1

An image of this type consists of a single load region in the load view and three execution regions in the execution view. The execution regions are placed contiguous in the memory map.

`-ro-base` *`address`* is used to specify the load and execution address of the region containing the RO output section.

The scatter load description equivalent to using `-ro-base 0x040000` is:

```
LR_1 0x040000 ; define the load region name as LR_1
               ; region starts at 0x040000
{              ; start of execution region descriptions
    ER_RO +0   ; first execution region is called ER_RO
               ; region starts at end of previous region
               ; since there was no previous region,
               ; address is 0x040000
    {
        *(+RO) ; all RO sections go into this region
               ; they are placed consecutively
    }
    ER_RW +0 ; second execution region is called ER_RW
             ; region starts at end of previous region
             ; address is 0x040000 + size of ER_RO region
    {
        *(+RW) ; all RW sections go into this region
               ; they are placed consecutively
    }
    ER_ZI +0 ; last execution region is called ER_ZI
             ; region starts at end of previous region at
             ; 0x040000 + size of ER_RO + size of ER_RW regions
    {
        *(+ZI) ; All ZI region are created here
               ; they are placed consecutively
    }
}
```

This description creates an image with one load region called LR_1, whose load address is `0x040000`.

The image has three execution regions, named `ER_RO`, `ER_RW` and `ER_ZI`, that contain the RO, RW and ZI output sections respectively. The execution address of ER_RO is `0x040000`. All three execution regions are placed contiguously in the memory map by using the `+`*`offset`* form of the base-designator for the execution region description. This allows an execution region to be placed immediately following the end of the preceding execution region.

Type 2

An image of this type consists of a single load region in the load view and three execution regions in the execution view. It is similar to images of type one except that the RW execution region is not placed contiguous with the RO execution region.

`-ro-base` *`address1`* is used to specify the load and execution address of the region containing the RO output section.

`-rw-base` *`address2`* is used to specify the execution address for the RW execution region.

The scatter load description equivalent to using `-ro-base 0x010000 -rw-base 0x040000` is:

```
LR_1 0x010000 ; define the load region name as LR_1
{
    ER_RO +0  ; first execution region is called ER_RO
              ; region starts at end of previous region
              ; since there was no previous region,
              ; address is 0x010000

    {
        *(+RO) ; all RO sections go into this region
               ; they are placed consecutively
    }
    ER_RW 0x040000 ; second execution region is called ER_RW
                   ; region starts at 0x040000
    {
        *(+RW) ; all RW sections go into this region
               ; they are placed consecutively
    }
    ER_ZI +0 ; last execution region is called ER_ZI
             ; address is 0x040000 + size of ER_RW region
    {
        *(+ZI) ; All ZI region are created here
    }
}
```

This description creates an image with one load region, named `LR_1`, with a load address of `0x010000`.

The image has three execution regions, named `ER_RO`, `ER_RW` and `ER_ZI`, that contain the RO,RW and ZI output sections respectively. The execution address of `ER_RO` is `0x010000`.

The `ER_RW` execution region is not contiguous with `ER_RO`, as its execution address is given by `0x040000`.

The `ER_ZI` execution region is placed immediately following the end of the preceding execution region, `ER_RW`.

Type 3

Type three images consists of two load regions in load view and three execution regions in execution view. It is similar to images of type two except that the single load region in type two is now split into two load regions

`-ro-base` *`address1`* is used to specify the load and execution address of the region containing the RO output section.

`-rw-base` *`address2`* is used to specify the load and execution address for the region containing the RW output section.

`-split` is used to split the default single load region (that contains the RO and RW output sections) into two load regions. One load region contains the RO output section and one contains the RW output section.

The scatter load description equivalent to using `-ro-base 0x010000 -rw-base 0x040000 -split` is:

```
LR_1 0x010000 ; first load region is at 0x010000
{
    ER_RO +0 ; address is 0x010000
    {
        *(+RO)
    }
}
LR_2 0x040000 ; second load region is at 0x040000
{
    ER_RW +0; address is 0x040000
    {
        *(+RW) ; all RW sections go into this region
               ; they are placed consecutively
    }
    ER_ZI +0 ; address is 0x040000 + size of ER_RW region
    {
        *(+ZI) ; all ZI sections go into this region
               ; they are placed consecutively
    }
}
```

This description creates an image with two load regions, named `LR_1` and `LR_2`, that have load addresses `0x010000` and `0x040000`.

The image has three execution regions, named `ER_RO`, `ER_RW` and `ER_ZI`, that contain the RO, RW, and ZI output sections respectively. The execution address of `ER_RO` is `0x010000`.

The ER_RW execution region is not contiguous with ER_RO and its execution address is given by 0x040000.

The ER_ZI execution region is placed immediately following the end of the preceding execution region, ER_RW.

Type 1 (using -ropi)

`-ro-base address` is used to specify the load and execution address of the region containing the RO output section.

`-ropi` marks the load and execution regions containing the RO output section as position-independent.

The scatter load description equivalent to using `-ro-base 0x010000 -ropi` is:

```
LR_1 0x010000 PI ; first load region is at 0x010000
{
    ER_RO +0    ; PI attribute is inherited from parent
                ; default execution address is 0x010000
                ; but the code can be moved
    {
        *(+RO) ; all the RO sections go here
    }
    ER_RW +0 ABSOLUTE ; PI attribute is overridden by ABSOLUTE
    {
        *(+RW) ; the RW sections are placed next
               ; they are not relocatable
    }
    ER_ZI +0    ; ER_ZI region placed after ER_RW region
    {
        *(+ZI)  ; all the ZI sections are placed here
    }
}
```

ER_RO, the RO execution region, inherits the PI attribute from the load region LR_1. The next execution region, ER_RW is marked as ABSOLUTE and uses the +*offset* form of base designator. This prevents ER_RW from inheriting the PI attribute from ER_RO.

Note

In the scatter load descriptions given in *Type 1* on page 6-51, *Type 2* on page 6-52, and *Type 3* on page 6-54, the load and execution regions did not have any explicitly specified attributes and were ABSOLUTE by default.

Similar scatter load descriptions can also be written to correspond to the usage of -ropi and -rwpi with Type 2 and Type 3 images.

Chapter 7
Toolkit Utilities

This chapter describes the software utilities provided with ADS. It contains the following sections:

7.1 Functions of the toolkit utilities

fromELF	The fromELF utility takes an ELF image file generated by armlink and produces image files in formats suited to ROM tools and to loading directly into memory. You can also use it to display various information about an ELF file or to generate text files containing the information.
armprof	The ARM profiler displays an execution profile of a program from a profile data file generated by an ARM debugger.
armar	The ARM librarian enables sets of ELF object files to be collected together and maintained in libraries. You can pass such a library to armlink in place of several ELF object files.
Flash downloader	The Flash downloader enables you to download binary images to the Flash memory of supported ARM development and evaluation boards.

-

7.2 The fromELF utility

fromELF is a utility that can translate *executable linkable format* (ELF) image files produced by armlink into other formats.

fromELF outputs the following image formats:

- Plain binary format
- *Extended Intellec Hex* (IHF) format
- Motorola 32-bit S-record format
- Intel Hex-32 format
- ELF format (resaves as ELF to, for example, convert a `-debug` ELF to a `-nodebug` ELF)
- *ARM Image Format* (AIF) family. The AIF family includes executable AIF, and non-executable AIF. This format is obsolete.

fromELF can also display information about the input file, for example disassembly output or symbol listings.

7.2.1 Image structure

fromELF can translate a file from ELF to other formats. It cannot alter the image structure or addresses. It is not possible to change a scatter-loaded ELF image into a non-scatter-loaded image in another format. Any structural or addressing information must be provided to armlink at link time.

7.2.2 fromELF command-line options

The `fromELF` command syntax is as follows:

```
fromelf  [-help] [-nodebug] [-vsn] [output_format]
[-output output_file] input_file
```

where:

`-help` This option shows help and usage information. If this option is specified, other command-line options are ignored. Calling fromELF without any parameters produces the same help information.

`-nodebug`	This option does not put debug information in the output files. This is the default for binary images. If `-nodebug` is specified, it affects all output formats. It overrides the `-text/g` option.
`-vsn`	This option displays fromELF version information.
`output_format`	This option selects the output file options. *output_format* can be one of:

`-bin`	Plain binary.
`-ihf`	Extended Intellec Hex format.
`-m32`	Motorola 32-Bit format (32 bit S-records).
`-i32`	Intel Hex-32 format.
`-elf`	ELF format (resaves as ELF). This can be used to convert a debug ELF image into a no-debug ELF image.
`-aifbin`	Non-executable AIF. This format is obsolete.
`-aif`	*ARM Image Format* (AIF) file. This format is obsolete.
`[-text]` *`text_categories`*	Image information in text format. You can decode an ELF image or ELF object file using this option. This is the default.

If *`output_file`* is not specified, the information is displayed on stdout.

If *`text_categories`* is not specified, the default is to output header information. If specified, *`categories`* consists of one or more of the following:

`c`	disassembles code
`d`	prints contents of the data sections
`g`	prints debug information
`r`	prints relocation information
`s`	prints the symbol table
`t`	prints the string table(s)

`v` prints detailed information on each segment and section header of the image

`z` prints the code and data sizes.

The category selectors can be specified as either:

- individual options, `-text -c -d`
- a single concatenated string, `-text -cd`
- category selectors only, `-c -d`
- multiple characters following a slash character, `-text/cd`

If an output format is not specified, the default output format of `-text` is used and the individual category selectors are recognized. If another output format is specified, the selectors are ignored.

`-output` *`output_file`*

This option specifies the name of the output file. Specifying the output file is optional with the `-text` output option and mandatory with all other outputs.

ELF images will contain multiple load regions if, for example, they are built with a scatter load description file that defines more than one load region.

If you use fromELF to convert an ELF image containing multiple load regions to a binary format using any of the `-bin`, `-ihf` `-m32`, or `-i32` options, fromELF creates an output directory named *`output_file`* and generates one binary output file for each load region in the input image. fromELF places the ouput files in the *`output_file`* directory.

`input_file` This option specifies the ELF file to be translated.

fromELF only accepts ARM-executable ELF files and ARM object ELF files (.o).

fromELF does not accept relocatable ELF files.

Examples

Example 7-1 Creating a plain binary file from an ELF file

```
fromelf -bin -o outfile.bin infile.axf
```

Example 7-2 Creating a plain binary file from an ELF file, text option ignored

```
fromelf -cs -bin -o outfile.bin infile.axf
```

Example 7-3 Converting a -debug ELF to -nodebug

```
fromelf -nodebug -elf -o outfile.ndb infile.axf
```

Example 7-4 Displaying a disassembly of an ELF file

```
fromelf -text -cs -o outfile.lst infile.axf

fromelf -text -c -s infile.axf

fromelf -cs -o outfile.txt infile.axf

fromelf -c -s infile.axf

fromelf -text/cs infile.axf
```

7.3 ARM profiler

The ARM profiler, armprof, displays an execution profile of a program from a profile data file generated by an ARM debugger. The profiler displays one of two types of execution profile depending on the amount of information present in the profile data:

- If only pc-sampling information is present, the profiler can display only a flat profile giving the percentage time spent in each function itself, excluding the time spent in any of its children.
- If function call count information is present, the profiler can display a *call graph* profile that shows approximations of the time spent in each function, the time accounted for by calls to all children of each function, and the time allocated to calls from different parents.

7.3.1 Profiler command-line options

A number of options are available to control the format and amount of detail present in the profiler output. The command syntax is as follows:

```
armprof [-parent|-noparent] [-child|-nochild] [-sort options]
prf_file
```

where:

`-parent`	This option displays information about the parents of each function in the profile listing. This gives approximate information about how much time is spent in each function servicing calls from each of its parents.
`-noparent`	This option turns off the parent listing.
`-child`	This option displays information about the children of each function. The profiler displays the approximate amount of time spent by each child performing services on behalf of the parent.
`-nochild`	This option turns off the child listing.

`-sort` *`options`*	This option sorts the profile information using one of the following options:

`cumulative`	sorts the output by the total time spent in each function and all its children.
`self`	sorts the output by the time spent in each function (excluding the time spent in its children).
`descendants`	Sorts the output by the time spent in all the children of a function but excluding time spent in the function itself.
`calls`	Sorts the output by the number of calls to each function in the listing.

`prf_file`	This option specifies the file containing the profile information.

By default, child functions are listed, but not parent functions, and the output is sorted by cumulative time.

Example

```
armprof -parent sort.prf
```

7.3.2 Sample output

The profiler output is split into a number of sections, each section separated by a line. Each section gives information on a single function. In a flat profile, one with no parent or child function information, each section is a single line.

The following shows sample sections for functions called `insert_sort` and `strcmp`.

```
Name                    cum%        self%   desc%       calls
-------------------------------------------------------
 main                               17.69%    60.06%          1
insert_sort             77.76%    17.69%    60.06%          1
 strcmp                             60.06%    0.00%      243432
-------------------------------------------------------
 qs_string_compare                   3.21%    0.00%       13021
 shell_sort                          3.46%    0.00%       14059
 insert_sort                        60.06%    0.00%      243432
strcmp                  66.75%    66.75%    0.00%      270512
-------------------------------------------------------
```

7.4 ARM librarian

The ARM librarian, armar, enables sets of ELF object files to be collected together and maintained in libraries. Such a library can then be passed to armlink in place of several object files. However, linking with an object library file does not necessarily produce the same results as linking with all the object files collected into the object library file. This is because armlink processes the input list and libraries differently:

- each object file in the input list appears in the output unconditionally, although unused areas are eliminated if the armlink `-remove` option is specified
- a member of library file is only included in the output if it is referred to by an object file or a previously processed library file.

For more information on how armlink processes its input files, refer to Chapter 6.

7.4.1 Librarian command-line options

The syntax of the armar command when used to extract files or library information is:

```
armar [ -help] [-C] [-entries] [-p] [-t] [-s] [-sizes] [-T] [-vsn]
[-v] [-via option_file] [-x] [-zs] [-zt] library [file_list]
```

The syntax when used to add or modify files in the library is:

```
armar [ -help] [-create] [-c] [-d] [-m] [-r] [-u] [-vsn] [-v]
[-via option_file] [ {-a|-b|-i}  pos_name] library [file_list]
```

where:

`-a`	This option places new files in *library* after the file *pos_name*.
`-b`	This option places new files in *library* before the file *pos_name*.
`-create`	This option creates a new library even if *library* already exists.
`-c`	This option suppresses the diagnostic message normally written to standard error when a library is created.
`-C`	This option instructs the librarian not to replace existing files with like-named files when performing extractions. This option is useful when `-T` is also used to prevent truncated file names from replacing files with the same prefix.
`-d`	This option deletes one or more files from *library*.
`-entries`	This option lists all entry points defined in *library*. The format for the listing is:

`ENTRY at offset` *`num`* `in section` *`name`* `of` *`member`*

`-help`	This option gives online details of the armar command.
`-i`	This option places new files in *library* before the file *pos_name* (equivalent to `-b`).
`-m`	This option moves files. If `-a`, `-b`, or `-i` with *pos_name* is specified, move files to the new position. Otherwise, move files to the end of library.
`-p`	This option prints the contents of files in *library* to standard output.
`-r`	This option replaces, or adds, files in *library*. If *library* does not exist, a new library file will be created and a diagnostic message will be written to standard error. If *file_list* is not specified and the library exists, the results are undefined. Files that replace existing files will not change the order of the library. If the `-u` option is used, then only those files with dates of modification later than the library files are replaced. If the `-a`, `-b`, or `-i` option is used, then *pos_name* must be present and specifies that new files are to be placed after (`-a`) or before (`-b` or `-i`) *pos_name*. Otherwise the new files are placed at the end.
`-t`	This option prints a table of contents of *library*. The files specified by *file_list* will be included in the written list. If *file_list* is not specified, all files in the library will be included in the order of the archive.
`-T`	This option allows file name truncation of extracted files whose library names are longer than the file system can support. By default, extracting a file with a name that is too long is an error. A diagnostic message will be written and the file will not be extracted.
`-u`	This option updates older files. When used with the `-r` option, files within *library* will be replaced only if the corresponding file has a modification time that is at least as new as the modification time of the file within library.

`-via` *`option_file`*

This option instructs the librarian to take options from *`option_file`*.

`-v`	This option gives verbose output. The output depends on what other options are used: `-d, -r or -x` Write a detailed file-by-file description of the library creation, the constituent files, and maintenance activity. `-p` Writes the name of the file to the standard output before writing the file itself to the standard output `-t` Includes a long listing of information about the files within the library. `-x` Prints the filename preceding each extraction.
`-sizes`	This option lists the `text`, `rodata`, `data` and `bss` size of each member in *library*. An example of the output is shown below:

```
text+rodata    data+bss     Member
516 + 0         0 + 256      app1.o
308 + 0         0 + 400      app2.o
0 + 24          0 + 0        apphdr.o
824 + 24        0 + 656      TOTAL
```

`-vsn`	This option prints the version number on standard error.
`-x`	This option extracts the files in *file_list* from *library*. The contents of *library* will not be changed. If no file operands are given, all files in *library* will be extracted. If the file name of a file extracted from the library is longer than that supported in the destination directory, the results are undefined.
`-zs`	This option shows the symbol table.
`-zt`	Lists member sizes and entry points in *library*. See `-sizes` and `-entries` for output format.
`library`	This is a path name of the library file.
`file_list`	This is a list of files to process. Each file is fully specified by its path and name. The path can be absolute, relative to drive and root, or relative to the current directory. Only the filename at the end of the path is used when comparing against the names of files in the library. If two or more path operands end with the same filename, the results are unspecified. You can use the wildcards `*` and `?` to specify files.

pos_name	This is the name of an existing library file to be used for relative positioning. This name must be supplied with options `-a`, `-b`, and `-i`.

Caution

The options `-a`, `-b`, `-C`, `-i`, `-m`, `-T`, `-u`, and `-v` are not required for normal operation.

7.4.2 Examples

Syntax examples are shown below:

Example 7-5 Create a new library and add all object files

```
armar -create mylib *.o
```

Example 7-6 List the table of contents

```
armar -t mylib
```

Example 7-7 List the symbol table

```
armar -zs mylib
```

Example 7-8 Add (or replace) files

```
armar -r my_lib obj1.o obj2.o obj3.o ...
armar -ru mylib k*.o
```

Example 7-9 Extract a group of files

```
armar -x my_lib k*.o
```

Example 7-10 Delete a group of files

```
armar -d my_lib sys_*
```

7.5 The Flash downloader

The Flash downloader is a utility provided with ADS, and accessible from the Windows debuggers AXD, ADU, ADW, and armsd. The Flash downloader is installed in:

- `install_directory\Bin\flash.li` a little-endian version
- `install_directory\Bin\flash.bi` a big-endian version

You can use the Flash downloader to program Flash memory on the board. This works only if Angel is running from RAM (the default) at the time, or if Multi-ICE or EmbeddedICE is being used rather than Angel. The correct version of Angel for the byte order of the board should be used.

The downloaded file must be in plain binary format. Refer to *The fromELF utility* on page 7-3 for information on converting an ELF format file to plain binary. The debugger downloads the Flash downloader into RAM on the target board. The Flash downloader then executes and fetches the code to be programmed into the Flash from the host using semihosting (see *ADS Debug Target Guide*).

The Flash downloader fails if it does not recognize the Flash memory being used. As supplied, the Flash downloader recognizes the two Flash devices supported by the ARM Development Board, the ATMEL AT29C040A (4 Mbit, 8-bit) and AT29C1024 (1 Mbit, 16-bit) Flash devices.

You must produce your own download utility if you are producing a different target system. You can, however, use the source for the Flash downloader as a basis. The source is provided in `install_directory\Examples\Flashload`.

For armsd, the Flash downloader should be passed the name of the file to be downloaded into Flash.

If a file is being downloaded, you are prompted for the sector from where the programming should start. If you are downloading Angel, it should be programmed into the start of the Flash, from sector 0.

If you have the Angel Ethernet Kit, the Flash downloader program can be used to override the default IP address and net mask used by Angel for Ethernet communication. To do this from armsd, pass the Flash download program the argument `-e`. The program prompts for the IP address and net mask. If you are using AXD, ADU or ADW, select the appropriate option from the menu.

7.5.1 Using the Flash downloader from AXD or ADW

Follow these steps to use the Flash downloader from AXD or ADW:

1. Select **Flash Download…** from the **File** menu. The Flash Download dialog is displayed (see Figure 7-1).

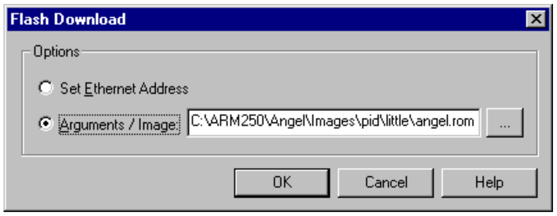


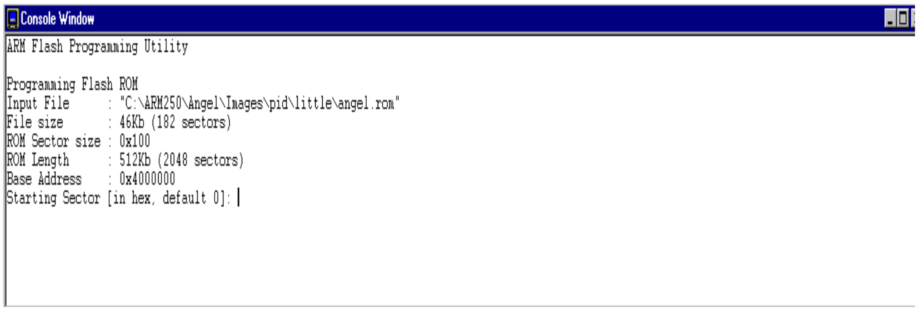
Figure 7-1 Flash Download dialog

2. Enter a pathname or click **Browse** to select a binary file to download.

   Note

   The pathname to the binary file must not contain spaces. If spaces are used, you must enclose the pathname in quotes.

3. Click **OK**. The Flash downloader reads the binary file and requests a start sector in the console window of the debugger (see Figure 7-2).



```
ARM Flash Programming Utility

Programming Flash ROM
Input File      : "C:\ARM250\Angel\Images\pid\little\angel.rom"
File size       : 46Kb (182 sectors)
ROM Sector size : 0x100
ROM Length      : 512Kb (2048 sectors)
Base Address    : 0x4000000
Starting Sector [in hex, default 0]: |
```

Figure 7-2 Console window

4. Enter a sector. In most cases the start sector will be zero and you can just press return. The console window displays the progress as the Flash is written.

7.5.2 Using the Flash downloader from armsd

To use the Flash downloader from the command line (assuming that you have a serial/parallel connection) write a batch file containing this command:

```
armsd -adp -port s,p -line 38400 -exec flash ROMname
```

where:

`flash`	Is the name of the Flash downloader, one of: • `install_directory\Bin\flash.li` for a little-endian system • `install_directory\Bin\flash.bi` for a big-endian system.
`ROMname`	Is the name of the binary file that you want to be programmed into Flash memory.

Note

The pathname to the binary file must not contain spaces. If spaces are used, you must enclose the pathname in quotes.

Execute the batch file to download to Flash. Enter the address to start writing from when prompted to do so.

Chapter 8
Floating-point Support

This chapter describes the ARM support for floating-point computations. It contains the following sections:

8.1 About floating-point support

ARM's floating-point environment is an implementation of the IEEE 754 standard for binary floating-point arithmetic. See *IEEE 754 arithmetic* on page 8-32 for details of the ARM implementation of the standard.

An ARM system may have:

- a *Vector Floating-Point* (VFP) coprocessor
- a *Floating-Point Accelerator* (FPA) coprocessor
- no floating-point hardware.

If you compile for a system with a hardware coprocessor (VFP or FPA), the compilers make use of it. If you compile for a system without a coprocessor, the compilers implement the calculations in software.

8.2 The software floating-point library, fplib

When programs are compiled to use a floating-point coprocessor, they perform basic floating-point arithmetic (for example addition and multiplication) by means of floating-point machine instructions for the target coprocessor. When programs are compiled to use software floating-point, there is no floating-point instruction set available, and so the ARM libraries have to provide a set of procedure calls to do floating-point arithmetic. These are the software floating-point library, fplib.

These routines have names like `_dadd` (add two **`double`**s) and `_fdiv` (divide two **`float`**s). The complete list is given in Table 8-1 on page 8-4, Table 8-2 on page 8-6, Table 8-3 on page 8-7 and Table 8-4 on page 8-8. User programs can call these routines directly. Even in environments with a coprocessor, the routines are provided, though they are typically only a few instructions long (as all they do is to execute the appropriate machine instruction).

All the fplib routines are called using a software floating-point variant of the calling standard. This means that floating-point arguments are passed and returned in integer registers. In the rest of the program, if the program is compiled for a coprocessor, floating-point data is passed in its floating-point registers.

So, for example, `_dadd` takes a **`double`** in registers a1 and a2, and another **`double`** in registers a3 and a4, and returns the sum in a1 and a2.

Note

For a **`double`** in registers a1 and a2, the register that holds the high 32 bits of the **`double`** depends on whether your program is little-endian or big-endian.

C programs are not required to handle the register allocation.

All the fplib routines are declared in the header file `rt_fp.h`. You can include this file if you want to call an fplib routine directly.

A complete list of the fplib routines is provided on the following pages.

8.2.1 Arithmetic on numbers in a particular format

The routines in Table 8-1 perform arithmetic on numbers in a particular format. Arguments and results are always in the same format.

Table 8-1 Arithmetic routines

Function	Argument types	Result type	Operation	Notes
`_fadd`	2 × **`float`**	**`float`**	Return x plus y	
`_fsub`	2 × **`float`**	**`float`**	Return x minus y	
`_frsb`	2 × **`float`**	**`float`**	Return y minus x	
`_fmul`	2 × **`float`**	**`float`**	Return x times y	
`_fdiv`	2 × **`float`**	**`float`**	Return x divided by y	
`_frdiv`	2 × **`float`**	**`float`**	Return y divided by x	
`_frem`	2 × **`float`**	**`float`**	Return remainder of x by y	a
`_frnd`	**`float`**	**`float`**	Return x rounded to an integer	b
`_fsqrt`	**`float`**	**`float`**	Return square root of x	
`_dadd`	2 × **`double`**	**`double`**	Return x plus y	
`_dsub`	2 × **`double`**	**`double`**	Return x minus y	
`_drsb`	2 × **`double`**	**`double`**	Return y minus x	
`_dmul`	2 × **`double`**	**`double`**	Return x times y	
`_ddiv`	2 × **`double`**	**`double`**	Return x divided by y	
`_drdiv`	2 × **`double`**	**`double`**	Return y divided by x	
`_drem`	2 × **`double`**	**`double`**	Return remainder of x by y	a
`_drnd`	**`double`**	**`double`**	Return x rounded to an integer	b
`_dsqrt`	**`double`**	**`double`**	Return square root of x	

Notes on Table 8-1:

a Describes functions that perform the IEEE 754 remainder operation. This is defined to take two numbers, *x* and *y*, and return a number *z* such that $z = x - n * y$, where *n* is an integer. In order to return an exactly correct result, *n* is chosen so that *z* is no bigger than half of *x* (so that *z* might be negative even if both *x* and *y* are positive). The IEEE 754 remainder function is therefore not the same as the operation performed by the C library function `fmod` (where *z* always has the same sign as *x*).

In the case where the above specification gives two acceptable choices of *n*, the even one is chosen. This behavior occurs independently of the current rounding mode.

b Describes functions that perform the IEEE 754 round-to-integer operation. This takes a number and rounds it to an integer (in accordance with the current rounding mode), but returns that integer in the floating-point number format rather than as a C **`int`** variable. To convert a number to an **`int`** variable, you should use the `_ffix` routines described in Table 8-2 on page 8-6.

8.2.2 Conversions between floats, doubles, and ints

The routines in Table 8-2 perform conversions between number formats, excluding long longs.

Table 8-2 Number format conversion routines

Function	Argument type	Result type	Notes
`_f2d`	**float**	**double**	
`_d2f`	**double**	**float**	
`_fflt`	**int**	**float**	
`_ffltu`	**unsigned int**	**float**	
`_dflt`	**int**	**double**	
`_dfltu`	**unsigned int**	**double**	
`_ffix`	**float**	**int**	a
`_ffix_r`	**float**	**int**	
`_ffixu`	**float**	**unsigned int**	a
`_ffixu_r`	**float**	**unsigned int**	
`_dfix`	**double**	**int**	a
`_dfix_r`	**double**	**int**	
`_dfixu`	**double**	**unsigned int**	a
`_dfixu_r`	**double**	**unsigned int**	

Note on Table 8-2:

a Always rounds toward zero, independently of the current rounding mode. This is because the C standard requires implicit conversions to integers to round this way, so it is convenient not to have to change the rounding mode in order to do so. Each function has a corresponding function with `_r` on the end of its name, that performs the same operation but rounds according to the current mode.

8.2.3 Conversions between long longs and other number formats

The routines in Table 8-3 perform conversions between **`long long`**s and other number formats.

Table 8-3 Conversion routines involving long long format

Function	Argument type	Result type	Notes
`_ll_sto_f`	**`long long`**	**`float`**	
`_ll_uto_f`	**`unsigned long long`**	**`float`**	
`_ll_sto_d`	**`long long`**	**`double`**	
`_ll_uto_d`	**`unsigned long long`**	**`double`**	
`_ll_sfrom_f`	**`float`**	**`long long`**	a
`_ll_sfrom_f_r`	**`float`**	**`long long`**	
`_ll_ufrom_f`	**`float`**	**`unsigned long long`**	a
`_ll_ufrom_f_r`	**`float`**	**`unsigned long long`**	
`_ll_sfrom_d`	**`double`**	**`long long`**	a
`_ll_sfrom_d_r`	**`double`**	**`long long`**	
`_ll_ufrom_d`	**`double`**	**`unsigned long long`**	a
`_ll_ufrom_d_r`	**`double`**	**`unsigned long long`**	

Note on Table 8-3:

a Always rounds toward zero, independently of the current rounding mode. This is because the C standard requires implicit conversions to integers to round this way, so it is convenient not to have to change the rounding mode in order to do so. Each function has a corresponding function with `_r` on the end of its name, that performs the same operation but rounds according to the current mode.

8.2.4 Floating-point comparisons

The routines in Table 8-4 perform comparisons between floating-point numbers.

Table 8-4 Floating-point comparison routines

Function	Argument types	Result type	Condition tested	Notes
`_fcmpeq`	2 × **`float`**	Flags, EQ/NE	x equal to y	a
`_fcmpge`	2 × **`float`**	Flags, HS/LO	x greater than or equal to y	a, b
`_fcmple`	2 × **`float`**	Flags, HI/LS	x less than or equal to y	a, b
`_feq`	2 × **`float`**	Boolean	x equal to y	
`_fneq`	2 × **`float`**	Boolean	x not equal to y	
`_fgeq`	2 × **`float`**	Boolean	x greater than or equal to y	b
`_fgr`	2 × **`float`**	Boolean	x greater than y	b
`_fleq`	2 × **`float`**	Boolean	x less than or equal to y	b
`_fls`	2 × **`float`**	Boolean	x less than y	b
`_dcmpeq`	2 × **`double`**	Flags, EQ/NE	x equal to y	a
`_dcmpge`	2 × **`double`**	Flags, HS/LO	x greater than or equal to y	a, b
`_dcmple`	2 × **`double`**	Flags, HI/LS	x less than or equal to y	a, b
`_deq`	2 × **`double`**	Boolean	x equal to y	
`_dneq`	2 × **`double`**	Boolean	x not equal to y	
`_dgeq`	2 × **`double`**	Boolean	x greater than or equal to y	b
`_dgr`	2 × **`double`**	Boolean	x greater than y	b
`_dleq`	2 × **`double`**	Boolean	x less than or equal to y	b
`_dls`	2 × **`double`**	Boolean	x less than y	b

Notes on Table 8-4:

a returns results in the ARM condition flags. This is efficient in assembly language, since you can directly follow a call to the function with a conditional instruction, but it means there is no way to use these functions from C. These functions are not declared in `rt_fp.h`.

b causes an Invalid Operation exception if either argument is a NaN, even a quiet NaN. Other functions only cause Invalid Operation if an argument is an SNaN. QNaNs return *not equal* when compared to anything, including other QNaNs (so comparing a QNaN to the same QNaN still returns *not equal*).

8.3 Controlling the floating-point environment

This section describes the functions you can use to control the ARM floating-point environment. With these functions, you can change the rounding mode, enable and disable trapping of exceptions, and install your own custom exception trap handlers.

ARM supplies several different interfaces to the floating-point environment, for compatibility and porting ease.

8.3.1 The __fp_status function

Previous versions of the ARM libraries implemented a function called `__fp_status`, that manipulated a status word in the floating-point environment. ARM still supports this function, for backwards compatibility. It is defined in `stdlib.h`.

`__fp_status` has the following prototype:

```
unsigned int __fp_status(unsigned int mask, unsigned int flags);
```

The function modifies the writable parts of the status word according to the parameters, and returns the previous value of the whole word.

The writable bits are modified by setting them to

```
new = (old & ~mask) ^ flags;
```

Four different operations can be performed on each bit of the status word, depending on the corresponding bits in mask and flags (see Table 8-5).

Table 8-5 Status word bit modification

Bit of mask	Bit of flags	Effect
0	0	Leave alone
0	1	Toggle
1	0	Set to 0
1	1	Set to 1

The layout of the status word as seen by `__fp_status` is shown in Figure 8-1.

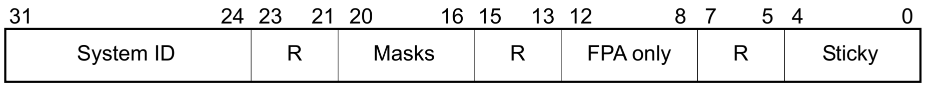


Figure 8-1 Floating-point status word layout

The fields in Figure 8-1 are as follows:

- Bits 0 to 4 (values `0x1` to `0x10`, respectively) are the sticky flags, or cumulative flags, for each exception. The sticky flag for an exception is set to 1 whenever that exception happens and is not trapped. Sticky flags are never cleared by the system, only by the user. The mapping of exceptions to bits is:
  - — bit 0 (`0x01`) is for the Invalid Operation exception
  - — bit 1 (`0x02`) is for the Divide by Zero exception
  - — bit 2 (`0x04`) is for the Overflow exception
  - — bit 3 (`0x08`) is for the Underflow exception
  - — bit 4 (`0x10`) is for the Inexact Result exception.
- Bits 8 to 12 (values `0x100` to `0x1000`) control various aspects of the FPA floating-point coprocessor. These bits are only writable when the FPA is being used.
- Bits 16 to 20 (values `0x10000` to `0x100000`) control whether each exception is trapped or not. If a bit is set to 1, the corresponding exception is trapped. If a bit is set to 0, the corresponding exception sets its sticky flag and return a plausible result, as described in *Exceptions* on page 8-37.
- Bits 24 to 31 contain the system ID that cannot be changed. It is set to `0x40` for software floating-point, to `0x80` or above for hardware floating-point, and to 0 or 1 if a hardware floating-point environment is being faked by an emulator.
- Bits marked R are reserved. They cannot be written to by the `__fp_status` call, and you should ignore anything you find in them.

The rounding mode cannot be changed with the `__fp_status` call.

As well as defining the `__fp_status` call itself, `stdlib.h` also defines some constants to be used for the arguments:

```
#define __fpsr_IXE  0x100000
#define __fpsr_UFE  0x80000
#define __fpsr_OFE  0x40000
#define __fpsr_DZE  0x20000
#define __fpsr_IOE  0x10000
```

```
#define __fpsr_IXC  0x10
#define __fpsr_UFC  0x8
#define __fpsr_OFC  0x4
#define __fpsr_DZC  0x2
#define __fpsr_IOC  0x1
```

For example, to trap the Invalid Operation exception and untrap all other exceptions, you would do:

```
__fp_status(_fpsr_IXE | _fpsr_UFE | _fpsr_OFE |
            _fpsr_DZE | _fpsr_IOE, _fpsr_IOE);
```

To untrap the Inexact Result exception:

```
__fp_status(_fpsr_IXE, 0);
```

To clear the Underflow sticky flag:

```
__fp_status(_fpsr_UFC, 0);
```

8.3.2 The __ieee_status function

ARM supports a second interface to the status word, similar to the `__fp_status` function, but the second interface sees the same status word in a different layout. This call is called `__ieee_status`, and it is generally the most efficient function to use for modifying the status word for VFP. (`__fp_status` is more efficient on FPA systems.) `__ieee_status` is defined in `fenv.h`.

Like `__fp_status`, `__ieee_status` has the prototype:

```
unsigned int __ieee_status(unsigned int mask,
                           unsigned int flags);
```

However, the layout of the status word as seen by `__ieee_status` is different from that seen by `__fp_status` (see Figure 8-2).

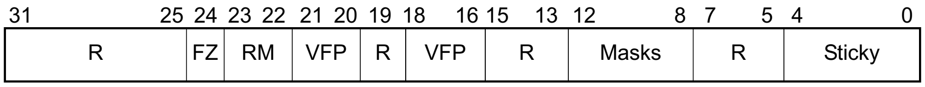


Figure 8-2 IEEE status word layout

The fields in Figure 8-2 are as follows:

- Bits 0 to 4 are the sticky flags, exactly as described in *The __fp_status function* on page 8-10.

- Bits 8 to 12 are the exception mask bits, exactly as described in *The __fp_status function* on page 8-10, but in a different place.
- Bits 16 to 18, and bits 20 and 21, are used by VFP hardware to control the VFP vector capability. The `__ieee_status` call does not let you modify these bits.
- Bits 22 and 23 control the rounding mode. See Table 8-6.

Table 8-6 Rounding mode control

Bits	Rounding mode
`00`	Round to nearest
`01`	Round up
`10`	Round down
`11`	Round toward zero

- Bit 24 enables FZ (Flush to Zero) mode if it is set. In FZ mode, denormals are forced to zero in order to speed up processing (since denormals can be difficult to work with and slow down floating-point systems). Setting this bit reduces accuracy but might increase speed.
- Bits marked R are reserved.

As well as defining the `__ieee_status` call itself, `fenv.h` also defines some constants to be used for the arguments:

```
#define FE_IEEE_FLUSHZERO          (0x01000000)
#define FE_IEEE_ROUND_TONEAREST    (0x00000000)
#define FE_IEEE_ROUND_UPWARD       (0x00400000)
#define FE_IEEE_ROUND_DOWNWARD     (0x00800000)
#define FE_IEEE_ROUND_TOWARDZERO   (0x00C00000)
#define FE_IEEE_ROUND_MASK         (0x00C00000)
#define FE_IEEE_MASK_INVALID       (0x00000100)
#define FE_IEEE_MASK_DIVBYZERO     (0x00000200)
#define FE_IEEE_MASK_OVERFLOW      (0x00000400)
#define FE_IEEE_MASK_UNDERFLOW     (0x00000800)
#define FE_IEEE_MASK_INEXACT       (0x00001000)
#define FE_IEEE_MASK_ALL_EXCEPT    (0x00001F00)
#define FE_IEEE_INVALID            (0x00000001)
#define FE_IEEE_DIVBYZERO          (0x00000002)
#define FE_IEEE_OVERFLOW           (0x00000004)
#define FE_IEEE_UNDERFLOW          (0x00000008)
#define FE_IEEE_INEXACT            (0x00000010)
#define FE_IEEE_ALL_EXCEPT         (0x0000001F)
```

For example, to set the rounding mode to round down, you would do:

```
__ieee_status(FE_IEEE_ROUND_MASK, FE_IEEE_ROUND_DOWNWARD);
```

To trap the Invalid Operation exception and untrap all other exceptions:

```
__ieee_status(FE_IEEE_MASK_ALL_EXCEPT, FE_IEEE_MASK_INVALID);
```

To untrap the Inexact Result exception:

```
__ieee_status(FE_IEEE_MASK_INEXACT, 0);
```

To clear the Underflow sticky flag:

```
__ieee_status(FE_IEEE_UNDERFLOW, 0);
```

8.3.3 Microsoft compatibility functions

The following three functions are implemented for compatibility with Microsoft products, to ease porting of floating-point code to the ARM architecture. They are defined in `float.h`.

The _controlfp function

The function `_controlfp` allows you to control exception traps and rounding modes:

```
unsigned int _controlfp(unsigned int new, unsigned int mask);
```

This function also modifies a control word using a mask to isolate the bits to modify. For every bit of `mask` that is zero, the corresponding control word bit is unchanged. For every bit of `mask` that is nonzero, the corresponding control word bit is set to the value of the corresponding bit of `new`. The return value is the previous state of the control word.

Note

This is not quite the same as the behavior of `__fp_status` and `__ieee_status`, where you can toggle a bit by setting a zero in the mask word and a one in the flags word.

The macros you can use to form the arguments to `_controlfp` are given in Table 8-7.

Table 8-7 _controlfp argument macros

Macro	Description
`_MCW_EM`	Mask containing all exception bits
`_EM_INVALID`	Bit describing the Invalid Operation exception
`_EM_ZERODIVIDE`	Bit describing the Divide by Zero exception
`_EM_OVERFLOW`	Bit describing the Overflow exception
`_EM_UNDERFLOW`	Bit describing the Underflow exception
`_EM_INEXACT`	Bit describing the Inexact Result exception
`_MCW_RC`	Mask for the rounding mode field
`_RC_CHOP`	Rounding mode value describing Round Toward Zero
`_RC_UP`	Rounding mode value describing Round Up
`_RC_DOWN`	Rounding mode value describing Round Down
`_RC_NEAR`	Rounding mode value describing Round To Nearest

Note

It is not guaranteed that the values of these macros will remain the same in future versions of ARM's products. To ensure that your code continues to work if the value changes in future releases, use the macro rather than its value.

For example, to set the rounding mode to round down, you would do:

```
_controlfp(_RC_DOWN, _MCW_RC);
```

To trap the Invalid Operation exception and untrap all other exceptions:

```
_controlfp(_EM_INVALID, _MCW_EM);
```

To untrap the Inexact Result exception:

```
_controlfp(0, _EM_INEXACT);
```

The _clearfp function

The function `_clearfp` clears all five exception sticky flags, and returns their previous value. The macros given in Table 8-7, for example `_EM_INVALID`, `_EM_ZERODIVIDE`, can be used to test bits of the returned result.

`_clearfp` has the following prototype:

```
unsigned _clearfp(void);
```

The _statusfp function

The function `_statusfp` returns the current value of the exception sticky flags. The macros given in Table 8-7 on page 8-15, for example `_EM_INVALID`, `_EM_ZERODIVIDE`, can be used to test bits of the returned result.

`_statusfp` has the following prototype:

```
unsigned _statusfp(void);
```

8.3.4 C9X-compatible functions

In addition to the above functions, ARM also supports a set of functions defined in the C9X draft standard. These functions are the only interface that allows you to install custom exception trap handlers with the ability to invent a return value. All the functions, types and macros in this section are defined in `fenv.h`.

C9X defines two data types, `fenv_t` and `fexcept_t`. The C9X draft standard does not define any details about these types, so for portable code you should treat them as opaque. ARM defines them to be structure types, for details see *ARM extensions to the C9X interface* on page 8-19.

The type `fenv_t` is defined to hold all the information about the current floating-point environment:

- the rounding mode
- the exception sticky flags
- whether each exception is masked
- what handlers are installed, if any.

The type `fexcept_t` is defined to hold all the information relevant to a given set of exceptions.

C9X also defines a macro for each rounding mode and each exception. The macros are as follows:

```
FE_DIVBYZERO
FE_INEXACT
FE_INVALID
FE_OVERFLOW
FE_UNDERFLOW
FE_ALL_EXCEPT
FE_DOWNWARD
FE_TONEAREST
FE_TOWARDZERO
FE_UPWARD
```

The exception macros are bit fields. The macro `FE_ALL_EXCEPT` is the bitwise OR of all of them.

Handling exception flags

C9X provides three functions to clear, test and raise exceptions:

```
void feclearexcept(int excepts);
int fetestexcept(int excepts);
void feraiseexcept(int excepts);
```

The `feclearexcept` function clears the sticky flags for the given exceptions. The `fetestexcept` function returns the bitwise OR of the sticky flags for the given exceptions (so that if the Overflow flag was set but the Underflow flag was not, then calling `fetestexcept(FE_OVERFLOW|FE_UNDERFLOW)` would return `FE_OVERFLOW`).

The `feraiseexcept` function raises the given exceptions, in unspecified order. If an exception trap is enabled for an exception raised this way, it is called.

C9X also provides functions to save and restore everything about a given exception. This includes the sticky flag, whether the exception is trapped, and the address of the trap handler, if any. These functions are:

```
void fegetexceptflag(fexcept_t *flagp, int excepts);
void fesetexceptflag(const fexcept_t *flagp, int excepts);
```

The `fegetexceptflag` function copies all the information relating to the given exceptions into the `fexcept_t` variable provided. The `fesetexceptflag` function copies all the information relating to the given exceptions from the `fexcept_t` variable into the current floating-point environment.

Note

`fesetexceptflag` can be used to set the sticky flag of a trapped exception to 1 without calling the trap handler, whereas `feraiseexcept` calls the trap handler for any trapped exception.

Handling rounding modes

C9X provides two functions for controlling rounding modes:

```
int fegetround(void);
int fesetround(int round);
```

The `fegetround` function returns the current rounding mode, as one of the macros defined above. The `fesetround` function sets the current rounding mode to the value provided. `fesetround` returns zero for success, or nonzero if its argument is not a valid rounding mode.

Saving the whole environment

C9X provides functions to save and restore the entire floating-point environment at once:

```
void fegetenv(fenv_t *envp);
void fesetenv(const fenv_t *envp);
```

The `fegetenv` function stores the current state of the floating-point environment into the `fenv_t` variable provided. The `fesetenv` function restores the environment from the variable provided.

Like `fesetexceptflag`, `fesetenv` does not call trap handlers when it sets the sticky flags for trapped exceptions.

Temporarily disabling exceptions

C9X provides two functions that allow you to avoid risking exception traps when executing code that might cause exceptions. This is useful when, for example, trapped exceptions are using ARM's default behavior. The default is to cause `SIGFPE` and terminate the application.

```
int feholdexcept(fenv_t *envp);
void feupdateenv(const fenv_t *envp);
```

-

The `feholdexcept` function saves the current floating-point environment in the `fenv_t` variable provided, sets all exceptions to be untrapped, and clears all the exception sticky flags. You can then execute code that might cause unwanted exceptions, and make sure the sticky flags for those exceptions are cleared. Then you can call `feupdateenv`. This restores any exception traps and calls them if necessary.

For example, suppose you have a function `frob()` that might cause the Underflow or Invalid Operation exceptions (assuming both exceptions are trapped). You are not interested in Underflow, but you want to know if an invalid operation is attempted. So you could do this:

```
fenv_t env;
feholdexcept(&env);
frob();
feclearexcept(FE_UNDERFLOW);
feupdateenv(&env);
```

Then, if the `frob()` function raises Underflow, it is cleared again by `feclearexcept`, and so no trap occurs when `feupdateenv` is called. However, if `frob()` raises Invalid Operation, the sticky flag is set when `feupdateenv` is called, and so the trap handler is invoked.

This mechanism is provided by C9X because C9X specifies no way to change exception trapping for individual exceptions. A better method is to use `__ieee_status` to disable the Underflow trap while leaving the Invalid Operation trap enabled. This has the advantage that the Invalid Operation trap handler is provided with all the information about the invalid operation (which operation was being performed on what data), and can invent a result for the operation. Using the C9X method, the Invalid Operation trap handler is called after the fact, receives no information about the cause of the exception, and is called too late to provide a substitute result.

8.3.5 ARM extensions to the C9X interface

ARM provides some extensions to the C9X interface, to allow it to do everything that the ARM floating-point environment is capable of. This includes trapping and untrapping individual exception types, and also installing custom trap handlers.

The types `fenv_t` and `fexcept_t` are not defined by C9X to be anything in particular. ARM defines them both to be the same structure type:

```
typedef struct {
    unsigned statusword;
    __ieee_handler_t invalid_handler;
    __ieee_handler_t divbyzero_handler;
    __ieee_handler_t overflow_handler;
    __ieee_handler_t underflow_handler;
    __ieee_handler_t inexact_handler;
} fenv_t, fexcept_t;
```

The members of the above structure are:

- `statusword` is the same status variable that the function `__ieee_status` sees, laid out in the same format (see *The __ieee_status function* on page 8-12).
- five function pointers giving the address of the trap handler for each exception. By default each is NULL. This means that if the exception is trapped then the default exception trap action happens. The default is to cause a `SIGFPE` signal.

Writing custom exception trap handlers

If you want to install a custom exception trap handler, you should declare it as a function like this:

```
__softfp__ieee_value_t myhandler(__ieee_value_t op1,
                                 __ieee_value_t op2,
                                 __ieee_edata_t edata);
```

The parameters to this function are:

- `op1` and `op2` are used to give the operands, or the intermediate result, for the operation that caused the exception:
  - — For the Invalid Operation and Divide by Zero exceptions, the original operands are supplied.
  - — For the Inexact Result exception, all that is supplied is the ordinary result that would have been returned anyway. This is provided in `op1`.
  - — For the Overflow exception, an intermediate result is provided. This result is calculated by working out what the operation would have returned if the exponent range had been big enough, and then adjusting the exponent so that it fits in the format. The exponent is adjusted by 192 (`0xC0`) in single precision, and by 1536 (`0x600`) in double precision.

    If Overflow happens when converting a **`double`** to a **`float`**, the result is supplied in **`double`** format, rounded to single precision, with the exponent biased by 192.

— For the Underflow exception, a similar intermediate result is produced, but the bias value is added to the exponent instead of being subtracted. The `edata` parameter also contains a flag to show whether the intermediate result has had to be rounded up, down, or not at all.

The type `__ieee_value_t` is defined as a union of all the possible types that an operand can be passed as:

```
typedef union {
    float f;
    float s;
    double d;
    int i;
    unsigned int ui;
    long long l;
    unsigned long long ul;
    struct { int word1, word2; } str;
} __ieee_value_t;
```

- `edata` contains flags that give details about the exception that occurred, and what operation was being performed. (The type `__ieee_edata_t` is a synonym for **`unsigned int`**.)
- The return value from the function is used as the result of the operation that caused the exception.

The flags contained in `edata` are:

- `edata & FE_EX_RDIR` is nonzero if the intermediate result in Underflow was rounded down, and 0 if it was rounded up or not rounded. (The difference between the last two is given in the Inexact Result bit.) This bit is meaningless for any other type of exception.
- `edata & FE_EX_`*`exception`* is nonzero if the given *`exception`* (`INVALID`, `DIVBYZERO`, `OVERFLOW`, `UNDERFLOW` or `INEXACT`) occurred. This allows you to:
  — use the same handler function for more than one exception type (the function can test these bits to tell what exception it is supposed to handle)
  — determine whether Overflow and Underflow intermediate results have been rounded or are exact.

  Since the `FE_EX_INEXACT` bit can be set in combination with either `FE_EX_OVERFLOW` or `FE_EX_UNDERFLOW`, you should determine the type of exception that actually occurred by testing Overflow and Underflow before testing Inexact.
- `edata & FE_EX_FLUSHZERO` is nonzero if the `FZ` bit was set when the operation was performed (see *The __ieee_status function* on page 8-12).

- `edata & FE_EX_ROUND_MASK` gives the rounding mode that applies to the operation. This is normally the same as the current rounding mode, unless the operation that caused the exception was a routine such as `_ffix`, that always rounds toward zero. The available rounding mode values are `FE_EX_ROUND_NEAREST`, `FE_EX_ROUND_PLUSINF`, `FE_EX_ROUND_MINUSINF` and `FE_EX_ROUND_ZERO`.

- `edata & FE_EX_INTYPE_MASK` gives the type of the operands to the function, as one of the type values shown in Table 8-8.

Table 8-8 FE_EX_INTYPE_MASK operand type flags

Flag	Operand type
`FE_EX_INTYPE_FLOAT`	**`float`**
`FE_EX_INTYPE_DOUBLE`	**`double`**
`FE_EX_INTYPE_INT`	**`int`**
`FE_EX_INTYPE_UINT`	**`unsigned int`**
`FE_EX_INTYPE_LONGLONG`	**`long long`**
`FE_EX_INTYPE_ULONGLONG`	**`unsigned long long`**

- `edata & FE_EX_OUTTYPE_MASK` gives the type of the operands to the function, as one of the type values shown in Table 8-9.

Table 8-9 FE_EX_OUTTYPE_MASK operand type flags

Flag	Operand type
`FE_EX_OUTTYPE_FLOAT`	**`float`**
`FE_EX_OUTTYPE_DOUBLE`	**`double`**
`FE_EX_OUTTYPE_INT`	**`int`**
`FE_EX_OUTTYPE_UINT`	**`unsigned int`**
`FE_EX_OUTTYPE_LONGLONG`	**`long long`**
`FE_EX_OUTTYPE_ULONGLONG`	**`unsigned long long`**

- `edata & FE_EX_FN_MASK` gives the nature of the operation that caused the exception, as one of the operation codes shown in Table 8-10.

Table 8-10 FE_EX_FN_MASK operation type flags

Flag	Operation type
`FE_EX_FN_ADD`	Addition
`FE_EX_FN_SUB`	Subtraction
`FE_EX_FN_MUL`	Multiplication
`FE_EX_FN_DIV`	Division
`FE_EX_FN_REM`	Remainder
`FE_EX_FN_RND`	Round to integer
`FE_EX_FN_SQRT`	Square root
`FE_EX_FN_CMP`	Compare.
`FE_EX_FN_CVT`	Convert between formats.
`FE_EX_FN_RAISE`	The exception was raised explicitly, by `feraiseexcept` or `feupdateenv`. In this case almost nothing in the `edata` word is valid.

When the operation is a comparison, the result should be returned as if it were an **`int`**, and should be one of the four values shown in Table 8-11.

Input and output types are the same for all operations except Compare and Convert.

Table 8-11 FE_EX_CMPRET_MASK comparison type flags

Flag	Comparison
`FE_EX_CMPRET_LESS`	op1 is less than op2
`FE_EX_CMPRET_EQUAL`	op1 is equal to op2
`FE_EX_CMPRET_GREATER`	op1 is greater than op2
`FE_EX_CMPRET_UNORDERED`	op1 and op2 are not comparable

Example 8-1 shows a custom exception handler. Suppose you are converting some Fortran code into C. The Fortran numerical standard requires 0 divided by 0 to be 1, whereas IEEE 754 defines 0 divided by 0 to be an Invalid Operation and so by default it returns a quiet NaN. The Fortran code is likely to rely on this behavior, and rather than modifying the code, it is probably easier to make 0 divided by 0 return 1.

A handler function that does this is shown in Example 8-1.

Example 8-1

```
__softfp__ ieee_value_t myhandler(__ieee_value_t op1, __ieee_value_t op2,
                                  __ieee_edata_t edata)
{
    __ieee_value_t ret;
    if ((edata & FE_EX_FN_MASK) == FE_EX_FN_DIV) {
        if ((edata & FE_EX_INTYPE_MASK) == FE_EX_INTYPE_FLOAT) {
            if (op1.f == 0.0 && op2.f == 0.0) {
                ret.f = 1.0;
                return ret;
            }
        }
        if ((edata & FE_EX_INTYPE_MASK) == FE_EX_INTYPE_DOUBLE) {
            if (op1.d == 0.0 && op2.d == 0.0) {
                ret.d = 1.0;
                return ret;
            }
        }
    }
    /* For all other invalid operations, raise SIGFPE as usual */
    raise(SIGFPE);
}
```

Instal the handler function as follows:

```
fenv_t env;
fegetenv(&env);
env.statusword |= FE_IEEE_MASK_INVALID;
env.invalid_handler = myhandler;
fesetenv(&env);
```

Once the handler is installed, dividing 0.0 by 0.0 returns 1.0.

Exception trap handling by signals

If an exception is trapped but the trap handler address is set to `NULL`, a default trap handler is used.

The default trap handler raises a `SIGFPE` signal. The default handler for `SIGFPE` prints an error message and terminates the program.

If you trap `SIGFPE`, you can declare your signal handler function to have a second parameter that tells you the type of floating-point exception that occurred. This feature is provided for compatibility with Microsoft products. The values are `_FPE_INVALID`, `_FPE_ZERODIVIDE`, `_FPE_OVERFLOW`, `_FPE_UNDERFLOW` and `_FPE_INEXACT`. They are defined in `float.h`. For example:

```
void sigfpe(int sig, int etype) {
    printf("SIGFPE (%s)\n",
           etype == _FPE_INVALID ? "Invalid Operation" :
           etype == _FPE_ZERODIVIDE ? "Divide by Zero" :
           etype == _FPE_OVERFLOW ? "Overflow" :
           etype == _FPE_UNDERFLOW ? "Underflow" :
           etype == _FPE_INEXACT ? "Inexact Result" :
           "Unknown");
}
signal(SIGFPE, (void(*)(int))sigfpe);
```

To generate your own `SIGFPE` signals with this extra information, you can call the function `__rt_raise` instead of the ANSI function raise. In Example 8-1 on page 8-24, instead of:

```
    raise(SIGFPE);
```

it is better to code:

```
    __rt_raise(SIGFPE, _FPE_INVALID);
```

`__rt_raise` is declared in `rt_misc.h`.

8.4 The math library, mathlib

Trigonometric functions in mathlib use range reduction to bring large arguments within the range 0 to 2π. ARM provides two different range reduction functions. One is accurate to one unit in the last place for *any* input values, but is larger and slower than the other. The other is reliable enough for almost all purposes and is faster and smaller.

The fast and small range reducer is used by default. To select the more accurate one, either:

- call the symbol `__use_accurate_range_reduction` from C
- `IMPORT` the symbol `__use_accurate_range_reduction` from assembly language

In addition to the functions defined by the ANSI C standard, mathlib provides the following functions:

- *Inverse hyperbolic functions (acosh, asinh, atanh)* on page 8-27
- *Cube root (cbrt)* on page 8-27
- *Copy sign (copysign)* on page 8-27
- *Error functions (erf, erfc)* on page 8-27
- *One less than exp(x) (expm1)* on page 8-28
- *Determine if a number is finite (finite)* on page 8-28
- *Gamma function (gamma, gamma_r)* on page 8-28
- *Hypotenuse function (hypot)* on page 8-28
- *Return the exponent of a number (ilogb)* on page 8-29
- *Determine if a number is a NaN (isnan)* on page 8-29
- *Bessel functions of the first kind (j0, j1, jn)* on page 8-29
- *The logarithm of the gamma function (lgamma, lgamma_r)* on page 8-29
- *Logarithm of one more than x (log1p)* on page 8-30
- *Return the exponent of a number (logb)* on page 8-30
- *Return the next representable number (nextafter)* on page 8-30
- *IEEE 754 remainder function (remainder)* on page 8-30
- *IEEE round-to-integer operation (rint)* on page 8-30
- *Scale a number by a power of two (scalb, scalbn)* on page 8-31
- *Return the fraction part of a number (significand)* on page 8-31
- *Bessel functions of the second kind (y0, y1, yn)* on page 8-31

8.4.1 Inverse hyperbolic functions (acosh, asinh, atanh)

```
double acosh(double x);
double asinh(double x);
double atanh(double x);
```

These functions are the inverses of the ANSI-required cosh, sinh and tanh:

- Since `cosh` is a symmetric function (that is, it returns the same value when applied to *x* or –*x*), `acosh` always has a choice of two return values, one positive and one negative. It chooses the positive result.
- `acosh` returns an `EDOM` error if called with an argument less than 1.0.
- `atanh` returns an `EDOM` error if called with an argument whose absolute value exceeds 1.0.

8.4.2 Cube root (cbrt)

```
double cbrt(double x);
```

This function returns the cube root of its argument.

8.4.3 Copy sign (copysign)

```
double copysign(double x, double y);
```

This function replaces the sign bit of *x* with the sign bit of *y*, and returns the result. It causes no errors or exceptions, even when applied to NaNs and infinities.

8.4.4 Error functions (erf, erfc)

```
double erf(double x);
double erfc(double x);
```

These functions compute the standard statistical error function, related to the Normal distribution:

- `erf` computes the ordinary error function of *x*.
- `erfc` computes one minus `erf(`*x*`)`. It is better to use `erfc(`*x*`)` than `1-erf(`*x*`)` when *x* is large, since the answer is more accurate.

8.4.5 One less than exp(*x*) (expm1)

```
double expm1(double x);
```

This function computes *e* to the power *x*, minus one. It is better to use `expm1(x)` than `exp(x)-1` if *x* is very near to zero, since `expm1` returns a more accurate value.

8.4.6 Determine if a number is finite (finite)

```
int finite(double x);
```

This function returns 1 if *x* is finite, and 0 if *x* is infinite or NaN. It does not cause any errors or exceptions.

8.4.7 Gamma function (gamma, gamma_r)

```
double gamma(double x);
double gamma_r(double x, int *);
```

These functions both compute the logarithm of the gamma function. They are synonyms for `lgamma` and `lgamma_r` (see *The logarithm of the gamma function (lgamma, lgamma_r)* on page 8-29).

Note

Despite their names, these functions compute the logarithm of the gamma function, not the gamma function itself.

8.4.8 Hypotenuse function (hypot)

```
double hypot(double x, double y);
```

This function computes the length of the hypotenuse of a right-angled triangle whose other two sides have length *x* and *y*. Equivalently, it computes the length of the vector (*x,y*) in Cartesian coordinates. Using `hypot(x,y)` is better than `sqrt(x*x+y*y)` because some values of *x* and *y* could cause $x * x + y * y$ to overflow even though its square root would not.

`hypot` returns an `ERANGE` error when the result does not fit in a **`double`**.

8.4.9 Return the exponent of a number (ilogb)

```
int ilogb(double x);
```

This function returns the exponent of *x*, without any bias, so `ilogb(1.0)` would return 0, and `ilogb(2.0)` would return 1, and so on.

When applied to 0, `ilogb` returns `-0x7FFFFFFF`. When applied to a NaN or an infinity, `ilogb` returns `+0x7FFFFFFF`. `ilogb` causes no exceptions or errors.

8.4.10 Determine if a number is a NaN (isnan)

```
int isnan(double x);
```

This function returns 1 if *x* is a NaN, and 0 otherwise. It causes no exceptions or errors.

8.4.11 Bessel functions of the first kind (j0, j1, jn)

```
double j0(double x);
double j1(double x);
double jn(int n, double x);
```

These functions compute Bessel functions of the first kind. `j0` and `j1` compute the functions of order 0 and 1 respectively. `jn` computes the function of order *n*.

If the absolute value of *x* exceeds π times 2^{52}, these functions return an `ERANGE` error, denoting total loss of significance in the result.

8.4.12 The logarithm of the gamma function (lgamma, lgamma_r)

```
double lgamma(double x);
double lgamma_r(double x, int *sign);
```

These functions compute the logarithm of the absolute value of the gamma function of *x*. The sign of the function is returned separately, so that the two can be used to compute the actual gamma function of *x*.

`lgamma` returns the sign of the gamma function of *x* in the global variable `signgam`. `lgamma_r` returns it in a user variable, whose address is passed in the `sign` parameter. The value, in either case, is either +1 or –1.

Both functions return an `ERANGE` error if the answer is too big to fit in a **`double`**.

Both functions return an `EDOM` error if *x* is zero or a negative integer.

8.4.13 Logarithm of one more than x (log1p)

```
double log1p(double x);
```

This function computes the natural logarithm of $x + 1$. Like `expm1`, it is better to use this function than `log(x+1)` because this function is more accurate when x is near zero.

8.4.14 Return the exponent of a number (logb)

```
double logb(double x);
```

This function is similar to `ilogb`, but returns its result as a **`double`**. It can therefore return special results in special cases.

- `logb(NaN)` is a quiet NaN.
- `logb(infinity)` is +infinity.
- `logb(0)` is –infinity, and causes a Divide by Zero exception.

`logb` is the same function as the `Logb` function described in the IEEE 754 Appendix.

8.4.15 Return the next representable number (nextafter)

```
double nextafter(double x, double y);
```

This function returns the next representable number after x, in the direction toward y. If x and y are equal, x is returned.

8.4.16 IEEE 754 remainder function (remainder)

```
double remainder(double x, double y);
```

This function is the IEEE 754 remainder operation. It is a synonym for `_drem` (see *Arithmetic on numbers in a particular format* on page 8-4).

8.4.17 IEEE round-to-integer operation (rint)

```
double rint(double x);
```

This function is the IEEE 754 round-to-integer operation. It is a synonym for `_drnd` (see *Arithmetic on numbers in a particular format* on page 8-4).

8.4.18 Scale a number by a power of two (scalb, scalbn)

```
double scalb(double x, double n);
double scalbn(double x, int n);
```

These functions return *x* times two to the power *n*. The difference between the functions is whether *n* is passed in as an **`int`** or as a **`double`**.

`scalb` is the same function as the `Scalb` function described in the IEEE 754 Appendix. Its behavior when *n* is not an integer is undefined.

8.4.19 Return the fraction part of a number (significand)

```
double significand(double x);
```

This function returns the fraction part of *x*, as a number between 1.0 and 2.0 (not including 2.0).

8.4.20 Bessel functions of the second kind (y0, y1, yn)

```
double y0(double x);
double y1(double x);
double yn(int, double);
```

These functions compute Bessel functions of the second kind. `y0` and `y1` compute the functions of order 0 and 1 respectively. `yn` computes the function of order *n*.

If *x* is positive and exceeds π times 2^{52}, these functions return an `ERANGE` error, denoting total loss of significance in the result.

8.5 IEEE 754 arithmetic

ARM's floating-point environment is an implementation of the IEEE 754 standard for binary floating-point arithmetic. This section contains a summary of the standard as it is implemented by ARM.

8.5.1 Basic data types

ARM floating-point values are stored in one of two data types, *single precision* and *double precision*. In this document these are called **`float`** and **`double`**. These are the corresponding C types.

Single precision

A **`float`** value is 32 bits wide. The structure is shown in Figure 8-3.

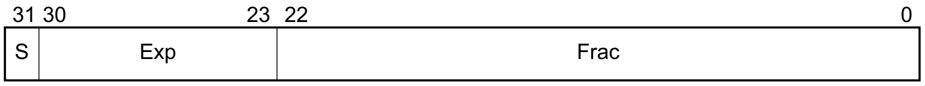


Figure 8-3 IEEE 754 single-precision floating-point format

The S field gives the sign of the number. It is 0 for positive, or 1 for negative.

The Exp field gives the exponent of the number, as a power of two. It is *biased* by `0x7F` (127), so that very small numbers have exponents near zero and very large numbers have exponents near `0xFF` (255). So, for example:

- if *Exp* = `0x7D` (125), the number is between 0.25 and 0.5 (not including 0.5)
- if *Exp* = `0x7E` (126), the number is between 0.5 and 1.0 (not including 1.0)
- if *Exp* = `0x7F` (127), the number is between 1.0 and 2.0 (not including 2.0)
- if *Exp* = `0x80` (128), the number is between 2.0 and 4.0 (not including 4.0)
- if *Exp* = `0x81` (129), the number is between 4.0 and 8.0 (not including 8.0).

The Frac field gives the fractional part of the number. It usually has an implicit 1 bit on the front, that is not stored in order to save space. So if *Exp* is `0x7F`, for example:

- if *Frac* = `00000000000000000000000` (binary), the number is 1.0
- if *Frac* = `10000000000000000000000` (binary), the number is 1.5
- if *Frac* = `01000000000000000000000` (binary), the number is 1.25
- if *Frac* = `11000000000000000000000` (binary), the number is 1.75.

So in general, the numeric value of a bit pattern in this format is given by the formula:

$(-1)^{S} * 2^{Exp(-0x7F)} * (1 + Frac * 2^{-23})$

Numbers stored in the above form are called *normalized* numbers.

The maximum and minimum exponent values, 0 and 255, are special cases. Exponent 255 is used to represent infinity, and store *Not a Number* (NaN) values. Infinity can occur as a result of dividing by zero, or as a result of computing a value that is too large to store in this format. NaN values are used for special purposes. Infinity is stored by setting Exp to 255 and Frac to all zeros. If Exp is 255 and Frac is nonzero, the bit pattern represents a NaN.

Exponent 0 is used to represent very small numbers in a special way. If Exp is zero, then the Frac field has no implicit 1 on the front. This means that the format can store 0.0, by setting both Exp and Frac to all 0 bits. It also means that numbers that are too small to store using Exp >= 1 are stored with less precision than the ordinary 23 bits. These are called *denormals*.

Double precision

A **`double`** value is 64 bits wide. Figure 8-4 shows its structure.

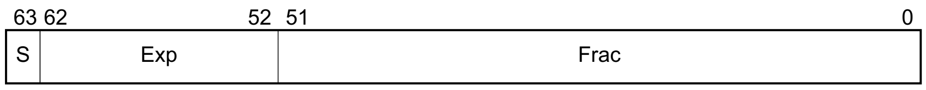


Figure 8-4 IEEE 754 double-precision floating-point format

As before, S is the sign, Exp the exponent, and Frac the fraction. Most of the discussion of **`float`** values remains true, except that:

- The Exp field is biased by `0x3FF` (1023) instead of `0x7F`, so numbers between 1.0 and 2.0 have an Exp field of `0x3FF`.
- The Exp value used to represent infinity and NaNs is `0x7FF` (2047) instead of `0xFF`.

Sample values

Some sample **`float`** and **`double`** bit patterns, together with their mathematical values, are given in Table 8-12 and Table 8-13.

Table 8-12 Sample single-precision floating-point values

Float value	S	Exp	Frac	Mathematical value	Notes
`0x3F800000`	0	`0x7F`	`000...000`	`1.0`	
`0xBF800000`	1	`0x7F`	`000...000`	`-1.0`	
`0x3F800001`	0	`0x7F`	`000...001`	`1.000 000 119`	a
`0x3F400000`	0	`0x7E`	`100...000`	`0.75`	
`0x00800000`	0	`0x01`	`000...000`	$1.18*10^{-38}$	b
`0x00000001`	0	`0x00`	`000...001`	$1.40*10^{-45}$	c
`0x7F7FFFFF`	0	`0xFE`	`111...111`	$3.40*10^{38}$	d
`0x7F800000`	0	`0xFF`	`000...000`	Plus infinity	
`0xFF800000`	1	`0xFF`	`000...000`	Minus infinity	
`0x00000000`	0	`0x00`	`000...000`	`0.0`	e
`0x7F800001`	0	`0xFF`	`000...001`	Signalling NaN	f
`0x7FC00000`	0	`0xFF`	`100...000`	Quiet NaN	f

The following notes apply to both Table 8-12, and Table 8-13 on page 8-35:

a The smallest representable number that can be seen to be greater than 1.0. The amount that it differs from 1.0 is known as the *machine epsilon*. This is 0.000 000 119 in **`float`**, and 0.000 000 000 000 000 222 in **`double`**. The machine epsilon gives a rough idea of the number of decimal places the format can keep track of. **`float`** can do six or seven places. **`double`** can do fifteen or sixteen.

b The smallest value that can be represented as a normalized number in each format. Numbers smaller than this can be stored as denormals, but are not held with as much precision.

c The smallest positive number that can be distinguished from zero. This is the absolute lower limit of the format.

d The largest finite number that can be stored. Attempting to increase this number by addition or multiplication causes overflow and generates infinity (in general).

e Zero. Strictly speaking, they show plus zero. Zero with a sign bit of 1, minus zero, is treated differently by some operations, although the comparison operations (for example `==` and `!=`) report that the two types of zero are equal.

f There are two types of NaNs, signalling NaNs and quiet NaNs. Quiet NaNs have a 1 in the first bit of Frac, and signalling NaNs have a zero there. The difference is that signalling NaNs cause an exception (see *Exceptions* on page 8-37) when used, whereas quiet NaNs do not.

Table 8-13 Sample double-precision floating-point values

Double value	S	Exp	Frac	Mathematical value	Notes
`0x3FF00000 00000000`	0	`0x3FF`	`000...000`	`1.0`	
`0xBFF00000 00000000`	1	`0x3FF`	`000...000`	`-1.0`	
`0x3FF00000 00000001`	0	`0x3FF`	`000...001`	`1.000 000 000 000 000 222`	a
`0x3FE80000 00000000`	0	`0x3FE`	`100...000`	`0.75`	
`0x00100000 00000000`	0	`0x001`	`000...000`	$2.23*10^{-308}$	b
`0x00000000 00000001`	0	`0x000`	`000...001`	$4.94*10^{-324}$	c
`0x7FEFFFFF FFFFFFFF`	0	`0x7FE`	`111...111`	$1.80*10^{308}$	d
`0x7FF00000 00000000`	0	`0x7FF`	`000...000`	Plus infinity	
`0xFFF00000 00000000`	1	`0x7FF`	`000...000`	Minus infinity	
`0x00000000 00000000`	0	`0x000`	`000...000`	`0.0`	e
`0x7FF00000 00000001`	0	`0x7FF`	`000...001`	Signalling NaN	f
`0x7FF80000 00000000`	0	`0x7FF`	`100...000`	Quiet NaN	f

-

8.5.2 Arithmetic and rounding

Arithmetic is generally performed by computing the result of an operation as if it were stored exactly (to infinite precision), and then rounding it to fit in the format. Apart from operations whose result already fits exactly into the format (such as adding 1.0 to 1.0), the correct answer is generally somewhere between two representable numbers in the format. The system then chooses one of these two numbers as the rounded result. It uses one of the following methods:

Round to nearest

The system chooses the nearer of the two possible outputs. If the correct answer is exactly half-way between the two, the system chooses the one where the least significant bit of Frac is zero. This behavior (round-to-even) prevents various undesirable effects.

This is the default mode when an application starts up. It is the only mode supported by the ordinary floating-point libraries. (Hardware floating-point environments and the enhanced floating-point libraries support all four modes.)

Round up, or **round toward plus infinity**

The system chooses the larger of the two possible outputs (that is, the one further from zero if they are positive, and the one closer to zero if they are negative).

Round down, or **round toward minus infinity**

The system chooses the smaller of the two possible outputs (that is, the one closer to zero if they are positive, and the one further from zero if they are negative).

Round toward zero, or **chop,** or **truncate**

The system chooses the output that is closer to zero, in all cases.

-

8.5.3 Exceptions

Floating-point arithmetic operations can run into various problems. For example, the result computed may be either too big or too small to fit into the format, or there may be no way to calculate the result (as in trying to take the square root of a negative number, or trying to divide zero by zero). These are known as exceptions, since they indicate unusual or exceptional situations.

ARM's floating-point environment can handle exceptions in more than one way.

Ignoring exceptions

The system invents a plausible result for the operation and returns that. For example, the square root of a negative number can produce a NaN, and trying to compute a value too big to fit in the format can produce infinity. If an exception occurs and is ignored, a flag is set in the floating-point status word to tell you that something went wrong at some point in the past.

Trapping exceptions

This means that when an exception occurs, a piece of code called a trap handler is run. The system provides a default trap handler, that prints an error message and terminates the application. However, you can supply your own trap handlers, that can clean up the exceptional condition in whatever way you choose. Trap handlers can even supply a result to be returned from the operation.

For example, if you had an algorithm where it was convenient to assume that 0 divided by 0 was 1, you could supply a custom trap handler for the Invalid Operation exception, that spotted that particular case and substituted the answer you wanted.

Types of exception

ARM's floating-point environment recognizes five different types of exception:

- The Invalid Operation exception happens when there is no sensible result for an operation. This can happen for any of the following reasons:
  - performing any operation on a signalling NaN, except the simplest operations (copying and changing the sign)
  - adding plus infinity to minus infinity, or subtracting an infinity from itself
  - multiplying infinity by zero
  - dividing 0 by 0, or dividing infinity by infinity
  - taking the remainder from dividing anything by 0, or infinity by anything
  - taking the square root of a negative number (not including minus zero)
  - converting a floating-point number to an integer if the result does not fit
  - comparing two numbers if one of them is a NaN.
- If the Invalid Operation exception is not trapped, all the above operations return a quiet NaN, except for conversion to an integer, which returns zero (as there are no quiet NaNs in integers).
- The Divide by Zero exception happens if you divide a finite nonzero number by zero. (Dividing zero by zero gives an Invalid Operation exception. Dividing infinity by zero is valid and returns infinity.) If Divide by Zero is not trapped, the operation returns infinity.
- The Overflow exception happens when the result of an operation is too big to fit into the format. This happens, for example, if you add the largest representable number (marked d in Table 8-12 on page 8-34) to itself. If Overflow is not trapped, the operation returns infinity, or the largest finite number, depending on the rounding mode.
- The Underflow exception can happen when the result of an operation is too small to be represented as a normalized number (with Exp at least 1). The situations that cause Underflow depends on whether it is trapped or not:
  - If Underflow is trapped, it occurs whenever a result is too small to be represented as a normalized number.
  - If Underflow is not trapped, it only occurs if the result actually loses accuracy due to being so small. So, for example, dividing the **`float`** number `0x00800000` by 2 does not signal Underflow, because the result (`0x00400000`) is still just as accurate as it would be if *Exp* had a greater range. However, trying to multiply the **`float`** number `0x00000001` by 1.5 does signal Underflow.

(For readers familiar with the IEEE 754 specification, ARM's choice of implementation options are to detect tininess after rounding, and to detect loss of accuracy as a denormalization loss.)

If Underflow is not trapped, the result is rounded to one of the two nearest representable denormal numbers, according to the current rounding mode. The loss of precision is ignored and the system returns the best result it can.

— The Inexact Result exception happens whenever the result of an operation requires rounding. This would cause significant loss of speed if it had to be detected on every operation in software, so the ordinary floating-point libraries do not support the Inexact Result exception. The enhanced floating-point libraries, and hardware floating-point systems, all support Inexact Result.

If Inexact Result is not trapped, the system rounds the result in the usual way.

The flag for Inexact Result is also set by Overflow and Underflow if either one of those is not trapped.

All exceptions are untrapped by default.

Glossary

ADS	See *ARM Developer Suite*.
ADU	See *ARM Debugger for UNIX*.
ADW	See *ARM Debugger for Windows*.
ANSI	American National Standards Institute. An organization that specifies standards for, among other things, computer software.
Angel	Angel is a program that enables you to develop and debug applications running on ARM-based hardware. Angel can debug applications running in either *ARM state* or *Thumb state*.
AOF	ARM Object Format

API	Application Program Interface.
Architecture	The term used to identify a group of processors that have similar characteristics.
ARM Debugger for UNIX	*ARM Debugger for UNIX* (ADU) and *ARM Debugger for Windows* (ADW) are two versions of the same ARM debugger software, running under UNIX or Windows respectively. This debugger was issued originally as part of the *ARM Software Development Toolkit*. It is still fully supported and is now supplied as part of the *ARM Developer Suite*.
ARM Debugger for Windows	*ARM Debugger for Windows* (ADW) and *ARM Debugger for UNIX* (ADU) are two versions of the same ARM debugger software, running under Windows or UNIX respectively. This debugger was issued originally as part of the *ARM Software Development Toolkit*. It is still fully supported and is now supplied as part of the *ARM Developer Suite*.
ARM Developer Suite	A suite of applications, together with supporting documentation and examples, that enable you to write and debug applications for the ARM family of *RISC* processors.
ARM eXtended Debugger	The *ARM eXtended Debugger* (AXD) is the latest debugger software from ARM that enables you to make use of a debug agent in order to examine and control the execution of software running on a debug target. AXD is supplied in both Windows and UNIX versions.
ARMulator	ARMulator is an instruction set simulator. It is a collection of modules that simulate the instruction sets and architecture of various ARM processors.
armsd	The *ARM Symbolic Debugger* (armsd) is an interactive source-level debugger providing high-level debugging support for languages such as C, and low-level support for assembly language. It is a command-line debugger that runs on all supported platforms.
ATPCS	ARM and Thumb Procedure Call Standard defines how registers and the stack will be used for subroutine calls.
AXD	See *ARM eXtended Debugger*.
Big-endian	Memory organization where the least significant byte of a word is at a higher address than the most significant byte.
BNF	Backus Naur Format. Mathematical notation for defining logical structures.
Byte	A unit of memory storage consisting of eight bits.
Canonical Frame Address	In DWARF 2, this is an address on the stack specifying where the call frame of an interrupted function is located.
CFA	See *Canonical Frame Address*.

Char	A unit of storage for a single character. ARM designs use a byte to store a single character and an integer to store two to four characters.
Class	A C++ class involved in the image.
Coprocessor	An additional processor which is used for certain operations. Usually used for floating-point math calculations, signal processing, or memory management.
Current place	In compiler terminology, the directory which contains files to be included in the compilation process.
Debugger	An application that monitors and controls the execution of a second application. Usually used to find errors in the application program flow.
Double word	A 64-bit unit of information. Contents are taken as being an unsigned integer unless otherwise stated.
DWARF	Debug With Arbitrary Record Format
EC++	A variant of C++ designed to be used for embedded applications.
ELF	Executable Linkable Format
Environment	The actual hardware and operating system that an application will run on.
Execution view	The address of regions and sections after the image has been loaded into memory and started execution.
Flash memory	Non-volatile memory that is often used to hold application code.
Globals	Variables or functions with the image with global scope.
Global variables	Variables that are accessible to all code in the application. *See also* Local variables
Halfword	A 16-bit unit of information. Contents are taken as being an unsigned integer unless otherwise stated.
Heap	The portion of computer memory that can be used for creating new variables.
Host	A computer which provides data and other services to another computer.
ICE	In Circuit Emulator.
IDE	Integrated Development Environment (CodeWarrior).
Image	An executable file which has been loaded onto a processor for execution.

-

	A binary execution file loaded onto a processor and given a thread of execution. An image may have multiple threads. An image is related to the processor on which its default thread runs.
Inline	Functions that are repeated in code each time they are used rather than having a common subroutine. Assembler code placed within a C or C++ program. *See also* Output sections
Input section	Contains code or initialized data or describes a fragment of memory that must be set to zero before the application starts. *See also* Output sections
Interrupt	A change in the normal processing sequence of an application caused by, for example, an external signal.
Interworking	Producing an application that uses both ARM and Thumb code.
Library	A collection of assembler or compiler output objects grouped together into a single repository.
Linker	Software which produces a single image from one or more source assembler or compiler output objects.
Little-endian	Memory organization where the least significant byte of a word is at a lower address than the most significant byte.
Local variable	A variable that is only accessible to the subroutine that created it. *See also* Global variables
Load view	The address of regions and sections when the image has been loaded into memory but has not yet started execution.
Memory management unit	Hardware that controls caches and access permissions to blocks of memory, and translates virtual to physical addresses.
MMU	See *Memory Management Unit*.
Monitor	A control showing the data associated with a particular debugger/target object. These may consist of a single, simple GUI control such as an edit field or a more complex multi-control dialog implemented as an ActiveX.
Multi-ICE	Multi-processor in-circuit emulator. ARM registered trademark.
Output section	A contiguous sequence of input sections that have the same RO, RW, or ZI attributes. The sections are grouped together in larger fragments called regions. The regions will be grouped together into the final executable image.

	See also Region
PIC	Position Independent Code. *See also* ROPI
PID	Position Independent Data *or* the ARM Platform-Independent Development card. *See also* RWPI
PIE	Platform-Independent Evaluator card. (ARM product.)
Profiling	Accumulation of statistics during execution of a program being debugged, to measure performance or to determine critical areas of code. *Call-graph profiling* provides great detail but slows execution significantly. *Flat profiling* provides simpler statistics with less impact on exectution speed. For both types of profiling you can specify the time interval between statistics-collecting operations.
Program image	See Image.
Redirection	The process of sending default output to a different destination or receiving default input from a different source. This is commonly used to output text, that would otherwise be displayed on the computer screen, to a file.
Reentrancy	The ability of a subroutine to have more that one instance of the code active. Each instance of the subroutine call has its own copy of any required static data.
Remapping	Changing the address of physical memory or devices after the application has started executing. This is typically done to allow RAM to replace ROM once the initialization has been done.
Regions	In an Image, a region is a contiguous sequence of one to three output sections (RO, RW, and ZI).
Retargeting	The process of moving code designed for one execution environment to a new execution environment.
ROPI	Read Only Position Independent. Code and read-only data addresses can be changed at run-time.
RTOS	Real Time Operating System.
RWPI	Read Write Position Independent. Read/write data addresses can be changed at run-time.
Scatter loading	Assigning the address and grouping of code and data sections individually rather than using single large blocks.

Scope	The accessibility of a function or variable at a particular point in the application code. Symbols which have global scope are always accessible. Symbols with local or private scope are only accessible to code in the same subroutine or object.
Sections	A block of software code or data for an Image. *See also* Input sections
Semihosting	A mechanism whereby the target communicates I/O requests made in the application code to the host system, rather attempting to support the I/O itself.
Signal	An indication of abnormal processor operation.
Stack	The portion of computer memory that is used to record the address of code that calls a subroutine. The stack can also be used for parameters and temporary variables.
SWI	Software Interrupt. An instruction that causes the processor to call a programer-specified subroutine. Used by ARM to handle semihosting.
Target	The actual target processor, (real or simulated), on which the target application is running. The fundamental object in any debugging session. The basis of the debugging system. The environment in which the target software will run. It is essentially a collection of real or simulated processors.
Thread	A thread of execution on a processor. A context of execution on a processor. A thread is always related to a processor and may or may not be associated with an image.
Volatile	Memory addresses where the contents can change independently of the executing application are described as volatile. *See also* Memory mapped
Veneer	A small block of code used with subroutine calls when there is a requirement to change processor state or branch to an address that cannot be reached in the current processor state.
VFP	Vector Floating Point. A standard for floating-point coprocessors where several data values can be processed by a single instruction.
Watchpoint	A location within the image which will be monitored and which will cause execution to break when it changes.
Word	A 32-bit unit of information. Contents are taken as being an unsigned integer unless otherwise stated.

ZI	Zero Initialized. R/W memory used to hold variables that do not have an initial value. The memory is normally set to zero on reset.

Index

The items in this index are listed in alphabetical order, with symbols and numerics appearing at the end. The references given are to page numbers.