

RealView[®] Compilation Tools

Version 3.1

Compiler Reference Guide



RealView Compilation Tools

Compiler Reference Guide

Copyright © 2007, 2010 ARM Limited. All rights reserved.

Release Information

The following changes have been made to this book.

Change History			
Date	Issue	Confidentiality	Change
March 2007	A	Non-confidential	Release 3.1 for RVDS v3.1
April 2010	A	Non-confidential	Documentation update for Release 3.1 for RVDS v3.1

Proprietary Notice

Words and logos marked with ® or ™ are registered trademarks or trademarks owned by ARM Limited. Other brands and names mentioned herein may be the trademarks of their respective owners.

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

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

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

Where the term ARM is used it means “ARM or any of its subsidiaries as appropriate”.

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Product Status

The information in this document is final, that is for a developed product.

Web Address

<http://www.arm.com>

Contents

RealView Compilation Tools

Compiler Reference Guide

Preface

About this book	vi
Feedback	x

Chapter 1

Introduction

1.1 About the ARM compiler	1-2
1.2 Source language modes	1-3
1.3 Language extensions and language compliance	1-5
1.4 The C and C++ libraries	1-8

Chapter 2

Compiler Command-line Options

2.1 Command-line options	2-2
--------------------------------	-----

Chapter 3

Language Extensions

3.1 Preprocessor extensions	3-2
3.2 C99 language features available in C90	3-5
3.3 C99 language features available in C++ and C90	3-7
3.4 Standard C language extensions	3-10
3.5 Standard C++ language extensions	3-15
3.6 Standard C and standard C++ language extensions	3-19

3.7	GNU language extensions	3-25
Chapter 4	Compiler-specific Features	
4.1	Keywords and operators	4-2
4.2	__declspec attributes	4-24
4.3	Function attributes	4-29
4.4	Type attributes	4-37
4.5	Variable attributes	4-40
4.6	Pragmas	4-49
4.7	Instruction intrinsics	4-63
4.8	Compiler predefines	4-103
Chapter 5	C and C++ Implementation Details	
5.1	C and C++ implementation details	5-2
5.2	C++ implementation details	5-13
Appendix A	Via File Syntax	
A.1	Overview of via files	A-2
A.2	Syntax	A-3
Appendix B	Standard C Implementation Definition	
B.1	Implementation definition	B-2
B.2	Behaviors considered undefined by the ISO C Standard	B-9
Appendix C	Standard C++ Implementation Definition	
C.1	Integral conversion	C-2
C.2	Calling a pure virtual function	C-3
C.3	Major features of language support	C-4
C.4	Standard C++ library implementation definition	C-5
Appendix D	C and C++ Compiler Implementation Limits	
D.1	C++ ISO/IEC standard limits	D-2
D.2	Limits for integral numbers	D-4
D.3	Limits for floating-point numbers	D-5
Appendix E	Using NEON Support	
E.1	Introduction	E-2
E.2	Vector data types	E-3
E.3	Intrinsics	E-4

Preface

This preface introduces the *RealView Compilation Tools Compiler Reference Guide*. It contains the following sections:

- *About this book* on page vi
- *Feedback* on page x.

About this book

This book provides reference information for *RealView Compilation Tools* (RVCT), and describes the command-line options to the ARM compiler. The book also gives reference material on the ARM implementation of C and C++ in the compiler. For general information on using and controlling the ARM compiler, see the *RVCT Compiler User Guide*.

Intended audience

This book is written for all developers who are producing applications using RVCT. It assumes that you are an experienced software developer. See the *RealView Compilation Tools Essentials Guide* for an overview of the ARM development tools provided with RVCT.

Using this book

This book is organized into the following chapters and appendixes:

Chapter 1 *Introduction*

Read this chapter for an overview of the ARM compiler, the conformance standards and the C and C++ Libraries.

Chapter 2 *Compiler Command-line Options*

Read this chapter for a list of all command-line options accepted by the ARM compiler.

Chapter 3 *Language Extensions*

Read this chapter for a description of the language extensions provided by the ARM compiler, and for information on standards conformance and implementation details.

Chapter 4 *Compiler-specific Features*

Read this chapter for a detailed list of ARM specific keywords, operators, pragmas, intrinsic functions, macros and semihosting Supervisor Calls (SVCs).

Chapter 5 *C and C++ Implementation Details*

Read this chapter for a description of the language implementation details for the ARM compiler.

Appendix A *Via File Syntax*

Read this appendix for a description of the syntax for via files. You can use via files to specify command-line arguments to many ARM tools.

Appendix B *Standard C Implementation Definition*

Read this appendix for information on the ARM C implementation that relates directly to the ISO C requirements.

Appendix C *Standard C++ Implementation Definition*

Read this appendix for information on the ARM C++ implementation.

Appendix D *C and C++ Compiler Implementation Limits*

Read this appendix for implementation limits of C and C++ in the ARM compiler.

Appendix E *Using NEON Support*

Read this appendix for information on the NEON™ intrinsics supported in this release of RVCT.

This book assumes that the ARM software is installed in the default location. For example, on Windows this might be *volume:\Program Files\ARM*. This is assumed to be the location of *install_directory* when referring to path names, for example *install_directory\Documentation\...* You might have to change this if you have installed your ARM software in a different location.

Typographical conventions

The following typographical conventions are used in this book:

monospace Denotes text that can be entered at the keyboard, such as commands, file and program names, and source code.

monospace Denotes a permitted abbreviation for a command or option. The underlined text can be entered instead of the full command or option name.

monospace italic

Denotes arguments to commands and functions where the argument is to be replaced by a specific value.

monospace bold

Denotes language keywords when used outside example code.

- 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.

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 Limited periodically provides updates and corrections to its documentation. See <http://www.arm.com> for current errata sheets and addenda, and the ARM Frequently Asked Questions (FAQs).

ARM publications

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

- *RVCT Essentials Guide* (ARM DUI 0202)
- *RVCT Compiler User Guide* (ARM DUI 0205)
- *RVCT Libraries and Floating Point Support Guide* (ARM DUI 0349)
- *RVCT Linker and Utilities Guide* (ARM DUI 0206)
- *RVCT Assembler Guide* (ARM DUI 0204)
- *RVCT Developer Guide* (ARM DUI 0203)
- *RealView Development Suite Glossary* (ARM DUI 0324)
- *NEON Vectorizing Compiler Guide* (ARM DUI 0350).

For full information about the base standard, software interfaces, and standards supported by ARM, see *install_directory\Documentation\Specifications\...*

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

- *ARM6-M Architecture Reference Manual* (ARM DDI 0419)
- *ARM7-M Architecture Reference Manual* (ARM DDI 0403)
- *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition* (ARM DDI 0406)
- *ARM Architecture Reference Manual Advanced SIMD Extension and VFPv3 Supplement* (ARM DDI 0268)

- ARM datasheet or technical reference manual for your hardware device.

Other publications

The following publications provide information about the ETSI basic operations. They are all available from the telecommunications bureau of the *International Telecommunications Union* (ITU) at <http://www.itu.int>.

- ETSI Recommendation G.191: *Software tools for speech and audio coding standardization*
- *ITU-T Software Tool Library 2005 User's manual*, included as part of ETSI Recommendation G.191
- ETSI Recommendation G723.1: *Dual rate speech coder for multimedia communications transmitting at 5.3 and 6.3 kbit/s*
- ETSI Recommendation G.729: *Coding of speech at 8 kbit/s using conjugate-structure algebraic-code-excited linear prediction (CS-ACELP)*.

Publications providing information about TI compiler intrinsics are available from Texas Instruments at <http://www.ti.com>.

Feedback

ARM Limited welcomes feedback on both RealView Compilation Tools and the documentation.

Feedback on RealView Compilation Tools

If you have any problems with RVCT, contact your supplier. To help them provide a rapid and useful response, 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 standalone sample of code that reproduces the problem
- a clear explanation of what you expected to happen, and what actually happened
- the commands you used, including any command-line options
- sample output illustrating the problem
- the version string of the tools, including the version number and build numbers.

Feedback on this book

If you notice any errors or omissions in this book, 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 compiler provided with *RealView Compilation Tools* (RVCT). It describes the standards of conformance and gives an overview of the runtime libraries provided with RVCT. It contains the following sections:

- *About the ARM compiler* on page 1-2
- *Source language modes* on page 1-3
- *Language extensions and language compliance* on page 1-5
- *The C and C++ libraries* on page 1-8.

1.1 About the ARM compiler

The ARM compiler, `armcc`, enables you to compile your C and C++ code.

The compiler:

- Is an optimizing compiler. Command-line options enable you to control the level of optimization.
- Compiles:
 - ISO Standard C:1990 source
 - ISO Standard C:1999 source, except for complex numbers and wide I/O
 - ISO Standard C++:2003 sourceinto:
 - 32-bit ARM code
 - 16/32-bit Thumb-2 code
 - 16-bit Thumb code.
- Complies with the *Base Standard Application Binary Interface for the ARM Architecture* (BSABI). In particular, the compiler:
 - Generates output objects in ELF format.
 - Generates DWARF Debugging Standard Version 3 (DWARF 3) debug information. RVCT also contains support for DWARF 2 debug tables.See *ABI for the ARM Architecture compliance* on page 1-3 in the *Libraries Guide* for more information.
- Can generate an assembly language listing of the output code, and can interleave an assembly language listing with source code.

If you are upgrading to RVCT from a previous release or are new to RVCT, ensure that you read *RealView Compilation Tools Essentials Guide* for the latest information.

1.2 Source language modes

The ARM compiler has three distinct source language modes that you can use to compile different varieties of C and C++ source code. These are:

- ISO C90
- ISO C99
- ISO C++.

1.2.1 ISO C90

The ARM compiler compiles C as defined by the 1990 C standard and addenda, excepting wide I/O:

- ISO/IEC 9899:1990. The 1990 International Standard for C.
- ISO/IEC 9899 AM1. The 1995 Normative Addendum 1, adding international character support through `wchar.h` and `wtype.h`.

The ARM compiler also supports several extensions to ISO C90. See *Language extensions and language compliance* on page 1-5 for more information.

Throughout this document, the term:

C90	Means ISO C90, excepting wide I/O, together with the ARM extensions. Use the compiler option <code>--c90</code> to compile C90 code. This is the default.
Strict C90	Means C as defined by the 1990 C standard and addenda, excepting wide I/O.

See also

- `--c90` on page 2-13
- *Language extensions and language compliance* on page 1-5
- Appendix B *Standard C Implementation Definition*.

1.2.2 ISO C99

The ARM compiler compiles C as defined by the 1999 C standard and addenda, excepting complex numbers and wide I/O:

- ISO/IEC 9899:1999. The 1999 International Standard for C.

The ARM compiler also supports several extensions to ISO C99. See *Language extensions and language compliance* on page 1-5 for more information.

Throughout this document, the term:

C99 Means ISO C99, excepting complex numbers and wide I/O, together with the ARM and GNU extensions.

Use the compiler option `--c99` to compile C99 code.

Strict C99 Means C as defined by the 1999 C standard and addenda, excepting complex numbers and wide I/O.

Standard C Means strict C99.

C Means any of C90, strict C90, C99, and Standard C.

See also

- `--c99` on page 2-13
- *Language extensions and language compliance* on page 1-5
- Appendix B *Standard C Implementation Definition*.

1.2.3 ISO C++

The ARM compiler compiles C++ as defined by the 2003 standard, excepting wide streams and export templates:

- ISO/IEC 14822:2003. The 2003 International Standard for C++.

The ARM compiler also supports several extensions to ISO C++. See *Language extensions and language compliance* on page 1-5 for more information.

Throughout this document, the term:

strict C++ Means ISO C++, excepting wide streams and export templates.

Standard C++ Means strict C++.

C++ Means ISO C++, excepting wide streams and export templates, either with or without the ARM extensions.

Use the compiler option `--cpp` to compile C++ code.

See also

- `--cpp` on page 2-14
- *Language extensions and language compliance* on page 1-5
- Appendix C *Standard C++ Implementation Definition*.

1.3 Language extensions and language compliance

The compiler supports numerous extensions to its various source languages. It also provides several command-line options for controlling compliance with the available source languages.

1.3.1 Language extensions

The language extensions supported by the compiler are categorized as follows:

- C99 features** The compiler makes some language features of C99 available:
- as extensions to strict C90, for example, `//`-style comments
 - as extensions to both Standard C++ and strict C90, for example, **restrict** pointers.

For more information see:

- *C99 language features available in C90* on page 3-5
- *C99 language features available in C++ and C90* on page 3-7.

Standard C extensions

The compiler supports numerous extensions to strict C99, for example, function prototypes that override old-style non-prototype definitions. See *Standard C language extensions* on page 3-10 for more information.

These extensions to Standard C are also available in C90.

Standard C++ extensions

The compiler supports numerous extensions to strict C++, for example, qualified names in the declaration of class members. See *Standard C++ language extensions* on page 3-15 for more information.

These extensions are not available in either Standard C or C90.

Standard C and Standard C++ extensions

The compiler supports some extensions specific to strict C++ and strict C90, for example, anonymous classes, structures, and unions. See *Standard C and standard C++ language extensions* on page 3-19 for more information.

GNU extensions The compiler supports some extensions offered by the GNU compiler, for example, GNU-style extended lvalues and GNU builtin functions. For more information see:

- *Language compliance*
- *GNU language extensions* on page 3-25
- Chapter 4 *Compiler-specific Features*.

ARM-specific extensions

The compiler supports a range of extensions specific to the ARM compiler, for example, instruction intrinsics and other builtin functions. See Chapter 4 *Compiler-specific Features* for more information.

1.3.2 Language compliance

The compiler has several modes in which compliance to a source language is either enforced or relaxed:

Strict mode In strict mode the compiler enforces compliance with the language standard relevant to the source language. For example, the use of *//*-style comments results in an error when compiling strict C90.

To compile in strict mode, use the command-line option `--strict`.

GNU mode In GNU mode all the GNU compiler extensions to the relevant source language are available. For example, in GNU mode:

- case ranges in **switch** statements are available when the source language is any of C90, C99 or non-strict C++
- C99-style designated initializers are available when the source language is either C90 or non strict C++.

To compile in GNU mode, use the compiler option `--gnu`.

————— **Note** —————

Some GNU extensions are also available when you are in a non-strict mode.

Example

The following examples illustrate combining source language modes with language compliance modes:

- Compiling a `.cpp` file with the command-line option `--strict` compiles Standard C++

- Compiling a C source file with the command-line option `--gnu` compiles GNU mode C90
- Compiling a `.c` file with the command-line options `--strict` and `--gnu` is an error.

See also

- `--gnu` on page 2-44
- `--[no_]strict` on page 2-84
- *GNU language extensions* on page 3-25
- *File naming conventions* on page 2-11 in the *Compiler User Guide*.

1.4 The C and C++ libraries

RVCT provides the following runtime C and C++ libraries:

The ARM C libraries

The ARM C libraries provide standard C functions, and helper functions used by the C and C++ libraries. The C libraries also provide target-dependent functions that are used to implement the standard C library functions such as `printf` in a semihosted environment. The C libraries are structured so that you can redefine target-dependent functions in your own code to remove semihosting dependencies.

The ARM libraries comply with:

- the *C Library ABI for the ARM Architecture* (CLIBABI)
- the *C++ ABI for the ARM Architecture* (CPPABI).

See *ABI for the ARM Architecture compliance* on page 1-3 in the *Libraries Guide* for more information.

Rogue Wave Standard C++ Library version 2.02.03

The Rogue Wave Standard C++ Library, as supplied by Rogue Wave Software, Inc., provides standard C++ functions and objects such as `cout`. It includes data structures and algorithms known as the *Standard Template Library* (STL). The C++ libraries use the C libraries to provide target-specific support. The Rogue Wave Standard C++ Library is provided with C++ exceptions enabled.

For more information on the Rogue Wave libraries, see the Rogue Wave HTML documentation and the Rogue Wave web site at:
<http://www.roguewave.com>

Support libraries

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

The C and C++ libraries are provided as binaries only. There is a variant of the 1990 ISO Standard C library for each combination of major build options, such as the byte order of the target system, whether interworking is selected, and whether floating-point support is selected.

See Chapter 2 *The C and C++ Libraries* in the *Libraries Guide* for more information.

Chapter 2

Compiler Command-line Options

This chapter lists the command-line options accepted by the ARM compiler, `armcc`. It includes the following section:

- *Command-line options* on page 2-2.

2.1 Command-line options

This section lists the command-line options supported by the compiler in alphabetical order.

2.1.1 `-Aopt`

This option specifies command-line options to pass to the assembler when it is invoked by the compiler to assemble either `.s` input files or embedded assembly language functions.

Syntax

`-Aopt`

Where:

`opt` is a command-line option to pass to the assembler.

———— **Note** ————

Some compiler command-line options are passed to the assembler automatically whenever it is invoked by the compiler. For example, if the option `--cpu` is specified on the compiler command line, then this option is passed to the assembler whenever it is invoked to assemble `.s` files or embedded assembler.

To see the compiler command-line options passed by the compiler to the assembler, use the compiler command-line option `-A--show_cmdline`.

Example

```
armcc -A--predefine="NEWVERSION SETL {TRUE}" main.c
```

Restrictions

If an unsupported option is passed through using `-A`, an error is generated.

See also

- `--cpu=name` on page 2-15
- `-Lopt` on page 2-52
- `--show_cmdline` on page 2-81.

2.1.2 --[no_]alternative_tokens

This option enables or disables the recognition of alternative tokens in C and C++.

Default

The default is `--alternative_tokens`.

Usage

In C and C++, use this option to control recognition of the digraphs. In C++, use this option to control recognition of operator keywords, for example, **and** and **bitand**.

2.1.3 --[no_]anachronisms

This option enables or disables anachronisms in C++.

Mode

This option is effective only if the source language is C++.

Default

The default is `--no_anachronisms`.

Example

```
typedef enum { red, white, blue } tricolor;

inline tricolor operator++(tricolor c, int)
{
    int i = static_cast<int>(c) + 1;
    return static_cast<tricolor>(i);
}

void foo(void)
{
    tricolor c = red;

    c++; // okay
    ++c; // anachronism
}
```

Compiling this code with the option `--anachronisms` generates a warning message.

Compiling this code without the option `--anachronisms` generates an error message.

See also

- `--cpp` on page 2-14
- `--[no_]strict` on page 2-84
- `--strict_warnings` on page 2-85
- *Anachronisms* on page 5-14.

2.1.4 `--apcs=qualifer...qualifier`

This option controls interworking and position independence when generating code.

By specifying qualifiers to the `--apcs` command-line option, you can define the variant of the *Procedure Call Standard for the ARM architecture* (AAPCS) used by the compiler.

Syntax

`--apcs=qualifer...qualifier`

Where *qualifer...qualifier* denotes a list of qualifiers. There must be:

- at least one qualifier present
- no spaces separating individual qualifiers in the list.

Each instance of *qualifier* must be one of:

<code>/[no]interwork</code>	Generates code with or without ARM/Thumb™ interworking support. The default is <code>/nointerwork</code> .
<code>/[no]ropi</code>	Enables or disables the generation of <i>Read-Only Position-Independent</i> (ROPI) code. The default is <code>/noropi</code> . <code>/[no]pic</code> is an alias for <code>/[no]ropi</code> .
<code>/[no]rwpi</code>	Enables or disables the generation of <i>Read/Write Position-Independent</i> (RWPI) code. The default is <code>/norwpi</code> . <code>/[no]pid</code> is an alias for <code>/[no]rwpi</code> .
<code>/[no]fpic</code>	Enables or disables the generation of read-only position-independent code where relative address references are independent of the location where your program is loaded.

————— Note —————

You can alternatively specify multiple qualifiers. For example,
`--apcs=/nointerwork/noropi/norwpi` is equivalent to `--apcs=/nointerwork`
`--apcs=/noropi/norwpi`.

Default

If you do not specify an `--apcs` option, the compiler assumes `--apcs=/nointerwork/noropi/norwpi`.

Usage

`/[no]interwork`

By default, code is generated:

- without interworking support, that is `/nointerwork`, unless you specify a `--cpu` option that corresponds to architecture ARMv5T or later
- with interworking support, that is `/interwork`, for ARMv5T or later. This is because ARMv5T or later provides direct interworking support.

`/[no]ropi`

If you select the `/ropi` qualifier to generate ROPI code, the compiler:

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

————— **Note** —————

`--apcs=/ropi` is not supported when compiling C++.

`/[no]rwpi`

If you select the `/rwpi` qualifier to generate RWPI code, the compiler:

- addresses writable data using offsets from the static base register `sb`. This means that:
 - the base address of the RW data region can be fixed at runtime
 - data can have multiple instances
 - data can be, but does not have to be, position-independent.
- sets the PI attribute on read/write output sections.

————— **Note** —————

Because the `--lower_rwpi` option is the default, code that is not RWPI is automatically transformed into equivalent code that is RWPI. This static initialization is done at runtime by the C++ constructor mechanism, even for C.

/[no]fpic

If you select this option, the compiler:

- accesses all static data using PC-relative addressing
- accesses all imported or exported read-write data using a *Global Offset Table (GOT)* entry created by the linker
- accesses all read-only data relative to the PC.

You must compile your code with /fpic if it uses shared objects. This is because relative addressing is only implemented when your code makes use of System V shared libraries.

You do not need to compile with /fpic if you are building either a static image or static library.

The use of /fpic is supported when compiling C++. In this case, virtual table functions and typeinfo are placed in read-write areas so that they can be accessed relative to the location of the PC.

Restrictions

There are restrictions when you compile code with /ropi, or /rwpi, or /fpic.

/ropi

The main restrictions when compiling with /ropi are:

- The use of --apcs=/ropi is not supported when compiling C++.
- Some constructs that are legal C do not work when compiled for --apcs=/ropi. For example:

```
extern const int ci; // ro
const int *p2 = &ci; // this static initialization
                        // does not work with --apcs=/ropi
```

To enable such static initializations to work, compile your code using the --lower_ropi option. For example:

```
armcc --apcs=/ropi --lower_ropi
```

/rwpi

The main restrictions when compiling with /rwpi are:

- Some constructs that are legal C do not work when compiled for --apcs=/rwpi. For example:

```
int i; // rw
int *p1 = &i; // this static initialization
              // does not work with --apcs=/rwpi
              // --no_lower_rwpi
```

To enable such static initializations to work, compile your code using the --lower_rwpi option. For example:

```
armcc --apcs=/rwpi
```

Note

You do not have to specify `--lower_ropi`, because this is the default.

`/fpic`

The main restrictions when compiling with `/fpic` are:

- If you use `--apcs=/fpic`, the compiler exports only functions and data marked `__declspec(dllexport)`.
- If you use `--apcs=/fpic` and `--no_hide_all` on the same command line, the compiler ignores `__declspec(dllexport)` and exports all extern variables and functions. The compiler disables auto-inlining for exported functions.
- If you use `--apcs=/fpic` in GNU mode, you must also use `--no_hide_all`.

See also

- `--[no_]hide_all` on page 2-46
- `--[no_]lower_ropi` on page 2-59
- `--[no_]lower_rwp_i` on page 2-59
- `__declspec(dllexport)` on page 4-24
- *Writing reentrant and thread-safe code* on page 2-4 in the *Libraries Guide*
- *Veneer generation* on page 3-20 in the *Linker Guide*
- Chapter 6 *BPABI and System V Shared Libraries and Executables* in the *Linker Guide*
- *Procedure Call Standard for the ARM architecture* in `install_directory\Documentation\Specifications\....`

2.1.5 `--arm`

This option configures the compiler to target the ARM instruction set.

Note

This option is not relevant for Thumb-only processors such as Cortex-M3.

Default

This is the default option for targets supporting the ARM instruction set.

See also

- `--cpu=list` on page 2-15
- `--cpu=name` on page 2-15
- `--thumb` on page 2-86
- `#pragma arm` on page 4-50
- *Specifying the target processor or architecture* on page 2-22 in the *Compiler User Guide*.

2.1.6 `--asm`

This option instructs the compiler to write a listing to a file of the disassembly of the machine code generated by the compiler.

Object code is generated when this option is selected. The link step is also performed, unless the `-c` option is chosen.

———— **Note** —————

To produce a disassembly of the machine code generated by the compiler, without generating object code, select `-S` instead of `--asm`.

Usage

The action of `--asm`, and the full name of the disassembly file produced, depends on the combination of options used:

Table 2-1 Compiling with the `--asm` option

Compiler option	Action
<code>--asm</code>	Writes a listing to a file of the disassembly of the compiled source. The link step is also performed, unless the <code>-c</code> option is used. The disassembly is written to a text file whose name defaults to the name of the input file with the filename extension <code>.s</code> .
<code>--asm -c</code>	As for <code>--asm</code> , except that the link step is not performed.

Table 2-1 Compiling with the `--asm` option (continued)

Compiler option	Action
<code>--asm --interleave</code>	As for <code>--asm</code> , except that the source code is interleaved with the disassembly. The disassembly is written to a text file whose name defaults to the name of the input file with the filename extension <code>.txt</code> .
<code>--asm --multifile</code>	As for <code>--asm</code> , except that the compiler produces empty object files for the files merged into the main file.
<code>--asm -o filename</code>	As for <code>--asm</code> , except that the object file is named <i>filename</i> . The disassembly is written to the file <i>filename.s</i> . The name of the object file must not have the filename extension <code>.s</code> . If the filename extension of the object file is <code>.s</code> , the disassembly is written over the top of the object file. This might lead to unpredictable results.

See also

- `-c` on page 2-12
- `--interleave` on page 2-50
- `--[no_]multifile` on page 2-63
- `-o filename` on page 2-65
- `-S` on page 2-80
- *File naming conventions* on page 2-11 in the *Compiler User Guide*.

2.1.7 `--[no_]autoinline`

This option enables or disables automatic inlining of functions.

The compiler automatically inlines functions where it is sensible to do so. The `-Ospace` and `-Otime` options, together with some other factors such as function size, influence how the compiler automatically inlines functions.

Selecting `-Otime`, in combination with various other factors, increases the likelihood that functions are inlined.

Default

For optimization levels `-O0` and `-O1`, the default is `--no_autoinline`.

For optimization levels `-O2` and `-O3`, the default is `--autoinline`.

See also

- `--forceinline` on page 2-39
- `--[no_]inline` on page 2-49
- `-Onum` on page 2-66
- `-Ospace` on page 2-68
- `-Otime` on page 2-69.

2.1.8 `--bigend`

This option instructs the compiler to generate code for an ARM processor using big-endian memory.

ARMv6 supports two different big-endian modes:

BE8 ARMv6 Byte Invariant Addressing mode.

BE32 Legacy big-endian mode.

The selection of BE8 versus BE32 is specified at link time.

Default

The compiler assumes `--littleend` unless `--bigend` is explicitly specified.

See also

- `--littleend` on page 2-57
- *Endian support* on page 4-45 in the *Compiler User Guide*
- *Specifying Byte Addressing mode* on page 2-27 in the *Linker Guide*.

2.1.9 `--[no_]brief_diagnostics`

This option enables or disables the output of brief diagnostic messages by the compiler.

When enabled, the original source line is not displayed, and error message text is not wrapped if it is too long to fit on a single line.

Default

The default is `--no_brief_diagnostics`.

Example

```
/* main.c */
#include <stdio.h>
```

```
int main(void)
{
    printf("Hello, world\n");
    return 0;
}
```

Compiling this code with `--brief_diagnostics` produces a warning message.

See also

- `--diag_error=tag[,tag,...]` on page 2-26
- `--diag_remark=tag[,tag,...]` on page 2-27
- `--diag_style={arm|ide|gnu}` on page 2-27
- `--diag_suppress=tag[,tag,...]` on page 2-28
- `--diag_warning=tag[,tag,...]` on page 2-30
- `--errors=filename` on page 2-34
- `--remarks` on page 2-78
- `-W` on page 2-94
- `--[no_]wrap_diagnostics` on page 2-96
- Chapter 5 *Diagnostic Messages* in the *Compiler User Guide*.

2.1.10 `--bss_threshold=num`

This option controls the placement of small global ZI data items in sections. A *small global ZI data item* is an uninitialized data item that is eight bytes or less in size.

Syntax

`--bss_threshold=num`

Where:

<i>num</i>	is either:
0	place small global ZI data items in ZI data sections
8	place small global ZI data items in RW data sections.

Default

If you do not specify a `--bss_threshold` option, the compiler assumes `--bss_threshold=8`.

Usage

In the current version of RVCT, the compiler might place small global ZI data items in RW data sections as an optimization. In RVCT 2.0.1 and earlier, small global ZI data items were placed in ZI data sections by default.

Use this option to emulate the behavior of RVCT 2.0.1 and earlier with respect to the placement of small global ZI data items in ZI data sections.

Note

Selecting the option `--bss_threshold=0` instructs the compiler to place *all* small global ZI data items in the current compilation module in a ZI data section. To place specific variables in:

- a ZI data section, use `__attribute__((zero_init))`
 - a specific ZI data section, use a combination of `__attribute__((section))__` and `__attribute__((zero_init))`.
-

Example

```
int glob1, glob2; /* ZI (.bss) in RVCT 2.0.1 and earlier */
                 /* RW (.data) in RVCT 2.1 and later */
```

Compiling this code with `--bss_threshold=0` places `glob1` and `glob2` in a ZI data section.

See also

- `#pragma arm section [section_sort_list]` on page 4-50
- `__attribute__((section))` on page 4-44
- `__attribute__((zero_init))` on page 4-48.

2.1.11 -c

This option instructs the compiler to perform the compilation step, but not the link step.

Note

This option is different from the uppercase `-C` option.

Usage

The use of the `-c` option is recommended in projects with more than one source file.

See also

- `--asm` on page 2-8
- `--list` on page 2-55
- `-o filename` on page 2-65
- `-S` on page 2-80.

2.1.12 -C

This option instructs the compiler to retain comments in preprocessor output.

Choosing this option implicitly selects the option `-E`.

Note

This option is different from the lowercase `-c` option.

See also

- `-E` on page 2-33.

2.1.13 --c90

This option enables the compilation of C90 source code.

Default

This option is implicitly selected for files having a suffix of `.c`, `.ac`, or `.tc`.

See also

- `--c99`
- `--gnu` on page 2-44
- `--[no_]strict` on page 2-84
- *Source language modes* on page 1-3
- *File naming conventions* on page 2-11 in the *Compiler User Guide*.

2.1.14 --c99

This option enables the compilation of C99 source code.

See also

- `--c90`
- `--gnu` on page 2-44

- *--[no_]strict* on page 2-84
- *Source language modes* on page 1-3.

2.1.15 `--[no_]code_gen`

This option enables or disables the generation of object code.

When generation of object code is disabled, the compiler performs syntax-checking only, without creating an object file.

Default

The default is `--code_gen`.

2.1.16 `--[no_]compile_all_input`

This option enables or disables the suppression of filename extension processing.

When enabled, the compiler suppresses filename extension processing entirely, treating all input files as if they have the suffix `.c`.

Default

The default is `--no_compile_all_input`.

See also

- *File naming conventions* on page 2-11 in the *Compiler User Guide*.

2.1.17 `--cpp`

This option enables the compilation of C++ source code.

Default

This option is implicitly selected for files having a suffix of `.cpp`, `.cxx`, `.c++`, `.cc`, `.CC`, `.acpp`, or `.tcpp`.

See also

- *--[no_]anachronisms* on page 2-3
- *--c90* on page 2-13
- *--c99* on page 2-13
- *--gnu* on page 2-44

- `--[no_]strict` on page 2-84
- *Source language modes* on page 1-3.

2.1.18 `--cpu=list`

This option lists the supported architecture and processor names that can be used with the `--cpu=name` option.

See also

- `--cpu=name`.

2.1.19 `--cpu=name`

This option enables code generation for the selected ARM processor or architecture.

Syntax

`--cpu=name`

Where:

name is the name of a processor or architecture.

If *name* is the name of a processor, enter it as shown on ARM data sheets, for example, ARM7TDMI, ARM1176JZ-S, MPCore.

If *name* is the name of an architecture, it must belong to the list of architectures shown in Table 2-2.

Processor and architecture names are not case-sensitive.

Wildcard characters are not accepted.

Table 2-2 Supported ARM architectures

Architecture	Description
4	ARMv4 without Thumb
4T	ARMv4 with Thumb
5T	ARMv5 with Thumb and interworking
5TE	ARMv5 with Thumb, interworking, DSP multiply, and double-word instructions
5TEJ	ARMv5 with Thumb, interworking, DSP multiply, double-word instructions, and Jazelle® extensions ^a

Table 2-2 Supported ARM architectures (continued)

Architecture	Description
6	ARMv6 with Thumb, interworking, DSP multiply, double-word instructions, unaligned and mixed-endian support, Jazelle, and media extensions
6-M	ARMv6 micro-controller profile with Thumb only plus processor state instructions
6K	ARMv6 with SMP extensions
6S-M	ARMv6 micro-controller profile with Thumb only, plus processor state instructions and OS extensions
6T2	ARMv6 with Thumb-2
6Z	ARMv6 with Security Extensions
7	ARMv7 with Thumb-2 only and without hardware divide ^b
7-A	ARMv7 application profile supporting virtual MMU-based memory systems, with ARM, Thumb, Thumb-2, and Thumb-2EE instruction sets, DSP support, and 32-bit SIMD support
7-R	ARMv7 real-time profile with ARM, Thumb, Thumb-2, DSP support, and 32-bit SIMD support
7-M	ARMv7 micro-controller profile with Thumb-2 only and hardware divide

- a. The ARM compiler cannot generate Java bytecodes.
- b. ARM v7 is not a recognized ARM architecture. Rather, it is the common subset of architectures ARMv7-A, ARMv7-R and ARMv7-M.

Default

If you do not specify a `--cpu` option, the compiler assumes `--cpu=ARM7TDMI`.
To get a full list of CPU architectures and processors, use the `--cpu=list` option.

Usage

The following general points apply to processor and architecture options:

Processors

- Selecting the processor selects the appropriate architecture, *Floating-Point Unit* (FPU), and memory organization.

- The supported `--cpu` values include all current ARM product names or architecture versions.
Other ARM architecture-based processors, such as the Marvell Feroceon and the Intel XScale, are also supported.
- If you specify a processor for the `--cpu` option, the compiled code is optimized for that processor. This enables the compiler to use specific coprocessors or instruction scheduling for optimum performance.

Architectures

- 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=5TE` produces code that can be used by the ARM926EJ-S®.

FPU

- Some specifications of `--cpu` imply an `--fpu` selection. For example, when compiling with the `--arm` option, `--cpu=ARM1136JF-S` implies `--fpu=vfpv2`.

———— **Note** ————

Any *explicit* FPU, set with `--fpu` on the command line, overrides an *implicit* FPU.

- If no `--fpu` option is specified and no `--cpu` option is specified, `--fpu=softvfp` is used.
- If an architecture v7 processor with an FPU is selected, the default FPU type is `--fpu=vfpv3`.

ARM/Thumb

- Specifying a processor or architecture that supports Thumb instructions, such as `--cpu=ARM7TDMI`, does not make the compiler generate Thumb code. It only enables features of the processor to be used, such as long multiply. Use the `--thumb` option to generate Thumb code.

———— **Note** ————

Specifying the target processor or architecture might make the object code generated by the compiler incompatible with other ARM processors. For example, code compiled for architecture ARMv6 might not run on an ARM920T processor, if the compiled

code includes instructions specific to ARMv6. Therefore, you must choose the lowest common denominator processor suited to your purpose.

- If you are compiling code that is intended for mixed ARM/Thumb systems for processors that support ARMv4T or ARMv5T, then you must specify the interworking option `--apcs=/interwork`. By default, this is enabled for processors that support ARMv5T or above.
- If you compile for Thumb, that is with the `--thumb` option on the command line, the compiler compiles as much of the code as possible using the Thumb instruction set. However, the compiler might generate ARM code for some parts of the compilation. For example, if you are compiling code for a Thumb-1 processor and using VFP, any function containing floating-point operations is compiled for ARM.
- If you are compiling code for ARMv7-M, for example `--cpu=Cortex-M3`, you do not need to specify `--thumb` on the command line, because ARMv7-M supports Thumb-2 only.

Restrictions

You cannot specify both a processor and an architecture on the same command-line.

See also

- `--apcs=qualifer...qualifier` on page 2-4
- `--cpu=list` on page 2-15
- `--fpu=name` on page 2-42
- `--thumb` on page 2-86.

2.1.20 `--create_pch=filename`

This option instructs the compiler to create a *PreCompiled Header* (PCH) file with the specified filename.

This option takes precedence over all other PCH options.

Syntax

`--create_pch=filename`

Where:

filename is the name of the PCH file to be created.

See also

- `--pch` on page 2-70
- `--pch_dir=dir` on page 2-71
- `--[no_]pch_messages` on page 2-72
- `--[no_]pch_verbose` on page 2-72
- `--use_pch=filename` on page 2-89
- `#pragma hdrstop` on page 4-55
- `#pragma no_pch` on page 4-57
- *Precompiled header files* on page 2-16 in the *Compiler User Guide*.

2.1.21 `-Dname[(parm-list)][=def]`

This option defines the macro *name*.

Syntax

`-Dname[(parm-list)][=def]`

Where:

name Is the name of the macro to be defined.

parm-list Is an optional list of comma-separated macro parameters. By appending a macro parameter list to the macro name, you can define function-style macros.

The parameter list must be enclosed in parentheses. When specifying multiple parameters, do not include spaces between commas and parameter names in the list.

———— **Note** ————

Parentheses might need escaping on UNIX systems.

=def Is an optional macro definition.

If *=def* is omitted, the compiler defines *name* as the value 1.

To include characters recognized as tokens on the command line, enclose the macro definition in double quotes.

Usage

Specifying `-Dname` has the same effect as placing the text `#define name` at the head of each source file.

Restrictions

The compiler defines and undefines macros in the following order:

1. compiler predefined macros
2. macros defined explicitly, using `-Dname`
3. macros explicitly undefined, using `-Uname`.

Example

Specifying the option:

```
-DMAX(X,Y)="((X > Y) ? X : Y)"
```

on the command line is equivalent to defining the macro:

```
#define MAX(X, Y) ((X > Y) ? X : Y)
```

at the head of each source file.

See also

- `-C` on page 2-13
- `-E` on page 2-33
- `-Uname` on page 2-87
- *Compiler predefines* on page 4-103.

2.1.22 `--[no_]data_reorder`

This option enables or disables automatic reordering of top-level data items, for example global variables.

The compiler can save memory by eliminating wasted space between data items. However, `--data_reorder` can break legacy code, if the code makes invalid assumptions about ordering of data by the compiler.

The ISO C Standard does not guarantee data order, so you must avoid writing code that depends on any assumed ordering. If you require data ordering, place the data items into a structure.

Default

The default is `--data_reorder`.

2.1.23 `--[no_]debug`

This option enables or disables the generation of debug tables for the current compilation.

The compiler produces the same code regardless of whether `--debug` is used. The only difference is the existence of debug tables.

Default

The default is `--no_debug`.

Using `--debug` does not affect optimization settings. By default, using the `--debug` option alone is equivalent to:

```
--debug --dwarf3 --debug_macros
```

See also

- `--[no_]debug_macros`
- `--dwarf2` on page 2-33
- `--dwarf3` on page 2-33
- `-Onum` on page 2-66.

2.1.24 `--[no_]debug_macros`

This option enables or disables the generation of debug table entries for preprocessor macro definitions.

Usage

Using `--no_debug_macros` might reduce the size of the debug image.

This option must be used with the `--debug` option.

Default

The default is `--debug_macros`.

See also

- `--[no_]debug`.

2.1.25 --default_extension=ext

This option enables you to change the filename extension for object files from the default extension (.o) to an extension of your choice.

Syntax

```
--default_extension=ext
```

Where:

`ext` is the filename extension of your choice.

Example

The following example creates an object file called `test.obj`, instead of `test.o`:

```
armcc --default_extension=obj -c test.c
```

———— Note ————

The `-o filename` option overrides this. For example, the following command results in an object file named `test.o`:

```
armcc --default_extension=obj -o test.o -c test.c
```

2.1.26 --[no_]dep_name

This option enables or disables dependent name processing in C++.

The C++ standard states that lookup of names in templates occurs:

- at the time the template is parsed, if the name is non dependent
- at the time the template is parsed, or at the time the template is instantiated, if the name is dependent.

When the option `--no_dep_name` is selected, the lookup of dependent names in templates can occur only at the time the template is instantiated. That is, the lookup of dependent names at the time the template is parsed is disabled.

———— Note ————

The option `--no_dep_name` is provided only as a migration aid for legacy source code that does not conform to the C++ standard. Its use is not recommended.

Mode

This option is effective only if the source language is C++.

Default

The default is `--dep_name`.

Restrictions

The option `--dep_name` cannot be combined with the option `--no_parse_templates`, because parsing is done by default when dependent name processing is enabled.

Errors

When the options `--dep_name` and `--no_parse_templates` are combined, the compiler generates an error.

See also

- `--[no_]parse_templates` on page 2-70
- *Template instantiation* on page 5-15.

2.1.27 `--depend=filename`

This option instructs the compiler to write make file dependency lines to a file during compilation.

Syntax

`--depend=filename`

Where:

filename is the name of the dependency file to be output.

Restrictions

If you specify multiple source files on the command line then any `--depend` option is ignored. No dependency file is generated in this case.

Usage

The output file is suitable for use by a make utility. To change the output format to be compatible with UNIX make utilities, use the `--depend_format` option.

See also

- `--depend_format=string`
- `--list` on page 2-55
- `--md` on page 2-60.

2.1.28 `--depend_format=string`

This option changes the format of output dependency files, for compatibility with some UNIX make programs.

Syntax

`--depend_format=string`

Where *string* is one of:

<code>unix</code>	generate dependency file entries using UNIX-style path separators.
<code>unix_escaped</code>	is the same as <code>unix</code> , but escapes spaces with <code>\</code> .
<code>unix_quoted</code>	is the same as <code>unix</code> , but surrounds path names with double quotes.

Usage

<code>unix</code>	On Windows systems, <code>--depend_format=unix</code> forces the use of UNIX-style path names. That is, the UNIX-style path separator symbol <code>/</code> is used in place of <code>\</code>. On UNIX systems, <code>--depend_format=unix</code> has no effect.
<code>unix_escaped</code>	On Windows systems, <code>--depend_format=unix_escaped</code> forces unix-style path names, and escapes spaces with <code>\</code>. On UNIX systems, <code>--depend_format=unix_escaped</code> with escapes spaces with <code>\</code>.
<code>unix_quoted</code>	On Windows systems, <code>--depend_format=unix_quoted</code> forces unix-style path names and surrounds them with <code>"</code>. On UNIX systems, <code>--depend_format=unix_quoted</code> surrounds path names with <code>"</code>.

Default

If you do not specify a `--depend_format` option, then the format of output dependency files depends on your choice of operating system:

Windows On Windows systems, the default is to use either Windows-style paths or UNIX-style paths, whichever is given.

UNIX On UNIX systems, the default is `--depend_format=unix`.

Example

On a Windows system, compiling a file `main.c` containing the line:

```
#include "..\include\header files\common.h"
```

using the options `--depend=depend.txt --depend_format=unix_escaped` produces a dependency file `depend.txt` containing the entries:

```
main.axf: main.c
main.axf: ../include/header\ files/common.h
```

See also

- `--depend=filename` on page 2-23
- `--md` on page 2-60.

2.1.29 --[no_]depend_system_headers

This option enables or disables the output of system include dependency lines when generating make file dependency information using either the `-M` option or the `--md` option.

Default

The default is `--depend_system_headers`.

Example

```
/* hello.c */
#include <stdio.h>

int main(void)
{
    printf("Hello, world!\n");
    return 0;
}
```

Compiling this code with the option `-M` produces:

```
__image.axf: hello.c
__image.axf: ...\\include\\...\\stdio.h
```

Compiling this code with the options `-M --depend_system_headers` produces:

```
__image.axf: hello.c
```

See also

- `-M` on page 2-60
- `--md` on page 2-60.

2.1.30 `--diag_error=tag[, tag,...]`

This option sets the diagnostic messages that have the specified tags to Error severity.

————— Note —————

This option has the `#pragma` equivalent `#pragma diag_error`.

Syntax

```
--diag_error=tag[, tag,...]
```

Where:

`tag[, tag,...]` is a comma-separated list of diagnostic message numbers specifying the messages whose severities are to be changed.

At least one diagnostic message number must be specified.

When specifying multiple diagnostic message numbers, do not include spaces between commas and numbers in the list.

Usage

The severity of the following types of diagnostic messages can be changed:

- Messages with the number format `#nnnn-D`.
- Warning messages with the number format `CnnnnW`.

See also

- `--diag_remark=tag[, tag,...]` on page 2-27
- `--diag_suppress=tag[, tag,...]` on page 2-28
- `--diag_warning=tag[, tag,...]` on page 2-30

- *#pragma diag_error tag[,tag,...]* on page 4-52
- *Changing the severity of diagnostic messages* on page 5-5 in the *Compiler User Guide*.

2.1.31 --diag_remark=tag[, tag, ...]

This option sets the diagnostic messages that have the specified tags to Remark severity.

The --diag_remark option behaves analogously to --diag_errors, except that the compiler sets the diagnostic messages having the specified tags to Remark severity rather than Error severity.

Note

Remarks are not displayed by default. To see remark messages, use the compiler option --remarks.

Note

This option has the #pragma equivalent #pragma diag_remark.

Syntax

--diag_remark=tag[, tag, ...]

Where:

tag[, tag, ...] is a comma-separated list of diagnostic message numbers specifying the messages whose severities are to be changed.

See also

- *--diag_error=tag[,tag,...]* on page 2-26
- *--diag_suppress=tag[,tag,...]* on page 2-28
- *--diag_warning=tag[,tag,...]* on page 2-30
- *--remarks* on page 2-78
- *#pragma diag_remark tag[,tag,...]* on page 4-53
- *Changing the severity of diagnostic messages* on page 5-5 in the *Compiler User Guide*.

2.1.32 --diag_style={arm|ide|gnu}

This option specifies the style used to display diagnostic messages.

Syntax

`--diag_style=string`

Where *string* is one of:

arm	Display messages using the ARM compiler style.
ide	Include the line number and character count for any line that is in error. These values are displayed in parentheses.
gnu	Display messages in the format used by gcc.

Default

If you do not specify a `--diag_style` option, the compiler assumes `--diag_style=arm`.

Usage

Choosing the option `--diag_style=ide` implicitly selects the option `--brief_diagnostics`. Explicitly selecting `--no_brief_diagnostics` on the command line overrides the selection of `--brief_diagnostics` implied by `--diag_style=ide`.

Selecting either the option `--diag_style=arm` or the option `--diag_style=gnu` does not imply any selection of `--brief_diagnostics`.

See also

- `--diag_error=tag[,tag,...]` on page 2-26
- `--diag_remark=tag[,tag,...]` on page 2-27
- `--diag_suppress=tag[,tag,...]`
- `--diag_warning=tag[,tag,...]` on page 2-30
- *Changing the severity of diagnostic messages* on page 5-5 in the *Compiler User Guide*.

2.1.33 `--diag_suppress=tag[, tag, ...]`

This option disables diagnostic messages that have the specified tags.

The `--diag_suppress` option behaves analogously to `--diag_errors`, except that the compiler suppresses the diagnostic messages having the specified tags rather than setting them to have Error severity.

———— **Note** —————

This option has the `#pragma` equivalent `#pragma diag_suppress`.

Syntax

`--diag_suppress=tag[, tag,...]`

Where:

`tag[, tag,...]` is a comma-separated list of diagnostic message numbers specifying the messages to be suppressed.

See also

- `--diag_error=tag[,tag,...]` on page 2-26
- `--diag_remark=tag[,tag,...]` on page 2-27
- `--diag_warning=tag[,tag,...]` on page 2-30
- `#pragma diag_suppress tag[,tag,...]` on page 4-54
- *Suppressing diagnostic messages* on page 5-6 in the *Compiler User Guide*.

2.1.34 `--diag_suppress=optimizations`

This option suppresses diagnostic messages for high-level optimizations.

Default

By default, optimization messages have Remark severity. Specifying `--diag_suppress=optimizations` suppresses optimization messages.

Note

Use the `--remarks` option to see optimization messages having Remark severity.

Usage

The compiler performs certain high-level vector and scalar optimizations when compiling at the optimization level `-O3`, for example, loop unrolling. Use this option to suppress diagnostic messages relating to these high-level optimizations.

Example

```

/*int*/ factorial(int n)
{
    int result=1;

    while (n > 0)
        result *= n--;
}

```

```
    return result;
}
```

Compiling this code with the options `-O3 -Otime --remarks --diag_suppress=optimizations` suppresses optimization messages.

See also

- `--diag_suppress=tag[,tag,...]` on page 2-28
- `--diag_warning=optimizations` on page 2-31
- `-Onum` on page 2-66
- `-Otime` on page 2-69
- `--remarks` on page 2-78.

2.1.35 `--diag_warning=tag[,tag,...]`

This option sets the diagnostic messages that have the specified tags to Warning severity.

The `--diag_warning` option behaves analogously to `--diag_errors`, except that the compiler sets the diagnostic messages having the specified tags to Warning severity rather than Error severity.

———— Note —————

This option has the `#pragma` equivalent `#pragma diag_warning`.

Syntax

`--diag_warning=tag[,tag,...]`

Where:

`tag[,tag,...]` is a comma-separated list of diagnostic message numbers specifying the messages whose severities are to be changed.

See also

- `--diag_error=tag[,tag,...]` on page 2-26
- `--diag_remark=tag[,tag,...]` on page 2-27
- `--diag_suppress=tag[,tag,...]` on page 2-28
- `#pragma diag_warning tag[,tag,...]` on page 4-54
- *Changing the severity of diagnostic messages* on page 5-5 in the *Compiler User Guide*.

2.1.36 `--diag_warning=optimizations`

This option sets high-level optimization diagnostic messages to have Warning severity.

Default

By default, optimization messages have Remark severity.

Usage

The compiler performs certain high-level vector and scalar optimizations when compiling at the optimization level `-O3 -Otime`, for example, loop unrolling. Use this option to display diagnostic messages relating to these high-level optimizations.

Example

```
int factorial(int n)
{
    int result=1;

    while (n > 0)
        result *= n--;

    return result;
}
```

Compiling this code with the options `--vectorize --cpu=Cortex-A8 -O3 -Otime --diag_warning=optimizations` generates optimization warning messages.

See also

- `--diag_suppress=optimizations` on page 2-29
- `--diag_warning=tag[,tag,...]` on page 2-30
- `-Onum` on page 2-66
- `-Otime` on page 2-69.

2.1.37 `--[no_]dllexport_all`

This option enables you to control symbol visibility when building DLLs.

Default

The default is `--no_dllexport_all`.

Usage

Use the option `--dll_export` to mark all extern definitions as `__declspec(dllexport)`.

See also

- `--apcs=qualifer...qualifier` on page 2-4
- `__declspec(dllexport)` on page 4-24.

2.1.38 `--[no_]dllimport_runtime`

This option enables you to control symbol visibility when using the runtime library as a shared library.

Default

The default is `--no_dllimport_runtime`.

Usage

Use the option `--dll_import_runtime` to mark:

- all builtin symbols as `__declspec(dllimport)`
- RTTI generated in the `cpprt` runtime libraries for import
- any optimized `printf()` and `__hardfp_` functions for import, provided that the original function is marked as `__declspec(dllimport)`.

See also

- `--[no_]guiding_decls` on page 2-44
- `--[no_]rtti` on page 2-79
- `__declspec(dllimport)` on page 4-25.

2.1.39 `--[no_]dollar`

This option instructs the compiler to accept or reject dollar signs, \$, in identifiers.

Default

The default is `--dollar`, unless the option `--strict` is specified.

If the option `--strict` is specified, then the default is `--no_dollar`.

See also

- `--[no_]strict` on page 2-84.

2.1.40 `--dwarf2`

This option instructs the compiler to use DWARF 2 debug table format.

Default

The compiler assumes `--dwarf3` unless `--dwarf2` is explicitly specified.

See also

- `--dwarf3`.

2.1.41 `--dwarf3`

This option instructs the compiler to use DWARF 3 debug table format.

Default

The compiler assumes `--dwarf3` unless `--dwarf2` is explicitly specified.

See also

- `--dwarf2`.

2.1.42 `-E`

This option instructs the compiler to execute only the preprocessor step.

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.

You can also use the `-o` option to specify a file for the preprocessed output. By default, comments are stripped from the output. The preprocessor accepts source files with any extension, for example, `.o`, `.s`, and `.txt`.

Example

```
armcc -E source.c > raw.c
```

See also

- `-C` on page 2-13

- `-o filename` on page 2-65.

2.1.43 `--enum_is_int`

This option forces the size of all enumeration types to be at least four bytes.

This option is switched off by default and the smallest data type is used that can hold the values of all enumerators.

Note

The `--enum_is_int` option is not recommended for general use.

2.1.44 `--errors=filename`

This option redirects the output of diagnostic messages from `stderr` to the specified errors file.

Syntax

`--errors=filename`

Where:

filename is the name of the file to which errors are to be redirected.

Diagnostics that relate to problems with the command options are not redirected, for example, if you type an option name incorrectly. However, if you specify an invalid argument to an option, for example `--cpu=999`, the related diagnostic is redirected to the specified *filename*.

See also

- `--[no_]brief_diagnostics` on page 2-10
- `--diag_error=tag[,tag,...]` on page 2-26
- `--diag_remark=tag[,tag,...]` on page 2-27
- `--diag_style={arm|ide|gnu}` on page 2-27
- `--diag_suppress=tag[,tag,...]` on page 2-28
- `--diag_warning=tag[,tag,...]` on page 2-30
- `--remarks` on page 2-78
- `-W` on page 2-94
- `--[no_]wrap_diagnostics` on page 2-96
- Chapter 5 *Diagnostic Messages* in the *Compiler User Guide*.

2.1.45 `--[no_]exceptions`

This option enables or disables exception handling.

In C++, the `--exceptions` option enables the use of `throw` and `try/catch`, causes function exception specifications to be respected, and causes the compiler to emit unwinding tables to support exception propagation at runtime.

In C++, when the `--no_exceptions` option is specified, `throw` and `try/catch` are not permitted in source code. However, function exception specifications are still parsed, but most of their meaning is ignored.

In C, the behavior of code compiled with `--no_exceptions` is undefined if an exception is thrown through the compiled functions. You must use `--exceptions`, if you want exceptions to propagate correctly through C functions.

Default

The default is `--no_exceptions`.

See also

- `--[no_]exceptions_unwind`.

2.1.46 `--[no_]exceptions_unwind`

This option enables or disables function unwinding for exception-aware code. This option is only effective if `--exceptions` is enabled.

When you use `--no_exceptions_unwind` and `--exceptions` then no exception can propagate through the compiled functions. `std::terminate` is called instead.

Default

The default is `--exceptions_unwind`.

See also

- `--[no_]exceptions`
- *Function unwinding at runtime* on page 5-18.

2.1.47 `--[no_]export_all_vtbl`

This option controls how dynamic symbols are exported in C++.

Mode

This option is effective only if the source language is C++.

Default

The default is `--no_export_all_vtbl`.

Usage

Use the option `--export_all_vtbl` to export all virtual table functions and *RunTime Type Information* (RTTI) for classes with a key function. A *key function* is the first virtual function of a class, in declaration order, that is not inline, and is not pure virtual.

———— Note ————

You can disable export for specific classes by using `__declspec(notshared)`.

See also

- `__declspec(notshared)` on page 4-27.

2.1.48 `--[no_]export_defs_implicitly`

This option controls how dynamic symbols are exported.

Default

The default is `--no_export_defs_implicitly`.

Usage

Use the option `--export_defs_implicitly` to export definitions where the prototype is marked `__declspec(dllexport)`.

See also

- `__declspec(dllexport)` on page 4-25.

2.1.49 `--[no_]extended_initializers`

This option enables the use of extended constant initializers even when compiling with `--strict` or `--strict_warnings`.

When certain non-portable but widely supported constant initializers such as the cast of an address to an integral type are used, `--extended_initializers` causes the compiler to produce the same general warning concerning constant initializers that it normally produces in non-strict mode, rather than specific errors stating that the expression must have a constant value or have arithmetic type.

Default

The default is `--no_extended_initializers` when compiling with `--strict` or `--strict_warnings`.

The default is `--extended_initializers` when compiling in non-strict mode.

See also

- `--[no_]strict` on page 2-84
- `--strict_warnings` on page 2-85
- *Constant expressions* on page 3-10.

2.1.50 `--feedback=filename`

This option enables the efficient elimination of unused functions.

Syntax

`--feedback=filename`

Where:

filename is the feedback file created by a previous execution of the ARM linker.

Usage

You can perform multiple compilations using the same feedback file, or specify multiple source files in a single compilation. The feedback file identifies the source file that contains the unused functions in a comment, and is not used by the compiler. The compiler places each unused function identified in the feedback file into its own ELF section in the corresponding object file.

The feedback file contains information about a previous build. Because of this:

- The feedback file might be out of date. That is, a function previously identified as being unused might be used in the current source. The linker removes the code for an unused function only if it is not used in the current source.

Note

For this reason, linker feedback is a safe optimization, but there might be a small impact on code size.

- You have to do a full compile and link at least twice to get the maximum benefit from linker feedback. However, a single compile and link using feedback from a previous build is usually sufficient.

See also

- *--split_sections* on page 2-83
- *Using linker feedback* on page 2-25 in the *Compiler User Guide*.

2.1.51 --[no_]force_new_nothrow

This option controls the behavior of `new` expressions in C++.

The C++ standard states that only a no throw operator `new`, one that is declared with `throw`, is permitted to return `NULL` on failure. Any other operator `new` is never permitted to return `NULL` and the default operator `new` throws an exception on failure.

If you use `--force_new_nothrow`, the compiler treats expressions such as `new T(...args...)`, that use the global `::operator new` or `::operator new[]`, as if they are `new (std::nothrow) T(...args...)`.

`--force_new_nothrow` also causes any class-specific operator `new` or any overloaded global operator `new` to be treated as no throw.

Note

The option `--force_new_nothrow` is provided only as a migration aid for legacy source code that does not conform to the C++ standard. Its use is not recommended.

Mode

This option is effective only if the source language is C++.

Default

The default is `--no_force_new_nothrow`.

Example

```
struct S
{
    void* operator new(std::size_t);
    void* operator new[](std::size_t);
};
void *operator new(std::size_t, int);
```

With the `--force_new_nothrow` option in effect, this is treated as:

```
struct S
{
    void* operator new(std::size_t) throw();
    void* operator new[](std::size_t) throw();
};
void *operator new(std::size_t, int) throw();
```

See also

- *Using the `::operator new` function* on page 5-13.

2.1.52 --forceinline

This option forces functions marked `__inline` to be inlined.

If used, the compiler always attempts to inline those functions marked as `__inline`, if possible. The compiler attempts to inline the function, regardless of its characteristics. However, it does not inline a function if doing so causes problems. For example, a recursive function is inlined into itself only once.

See also

- *--[no_]autoinline* on page 2-9
- *--[no_]inline* on page 2-49
- *__forceinline* on page 4-6
- *__inline* on page 4-9
- *Managing inlining* on page 4-20 in the *Compiler User Guide*.

2.1.53 --fpmode=mode1

This option specifies the floating-point conformance, and sets library attributes and floating-point optimizations.

Syntax

```
--fpmode=mode1
```

Where *model* is one of:

ieee_full All facilities, operations, and representations guaranteed by the IEEE standard are available in single and double-precision. Modes of operation can be selected dynamically at runtime.

This defines the symbols:

```
__FP_IEEE
__FP_FENV_EXCEPTIONS
__FP_FENV_ROUNDING
__FP_INEXACT_EXCEPTION
```

ieee_fixed

IEEE standard with round-to-nearest and no inexact exceptions.

This defines the symbols:

```
__FP_IEEE
__FP_FENV_EXCEPTIONS
```

ieee_no_fenv

IEEE standard with round-to-nearest and no exceptions. This mode is stateless and is compatible with the Java floating-point arithmetic model.

This defines the symbol `__FP_IEEE`.

std IEEE finite values with denormals flushed to zero, round-to-nearest, and no exceptions. This is compatible with standard C and C++ and is the default option.

Normal finite values are as predicted by the IEEE standard. However:

- NaNs and infinities might not be produced in all circumstances defined by the IEEE model. When they are produced, they might not have the same sign.
- The sign of zero might not be that predicted by the IEEE model.

fast Perform more aggressive floating-point optimizations that might cause a small loss of accuracy to provide a significant performance increase. This option defines the symbol `__FP_FAST`.

This option results in behavior that is not fully compliant with the ISO C or C++ standard. However, numerically robust floating-point programs are expected to behave correctly.

A number of transformations might be performed, including:

- Double-precision math functions might be converted to single precision equivalents if all floating-point arguments can be exactly represented as single precision values, and the result is immediately converted to a single-precision value.

This transformation is only performed when the selected library contains the single-precision equivalent functions, for example, when the selected library is `rvct` or `aeabi_glibc`.

For example:

```
float f(float a)
{
    return sqrt(a);
}
```

is transformed to

```
float f(float a)
{
    return sqrtf(a);
}.
```

- Double-precision floating-point expressions that are narrowed to single-precision are evaluated in single-precision when it is beneficial to do so. For example, `float y = (float)(x + 1.0)` is evaluated as `float y = (float)x + 1.0f`.
- Division by a floating-point constant is replaced by multiplication with the inverse. For example, `x / 3.0` is evaluated as `x * (1.0 / 3.0)`.
- It is not guaranteed that the value of `errno` is compliant with the ISO C or C++ standard after math functions have been called. This enables the compiler to inline the VFP square root instructions in place of calls to `sqrt()` or `sqrtf()`.

See also

- *The `--fpmode` option on page 4-34 in the Compiler User Guide*
- *ARM Application Note 133 - Using VFP with RVDS in `install_directory\RVDS\Examples\...\vfpsupport`.*

2.1.54 `--fpu=list`

This option lists the supported FPU architecture names that you can use with the `--fpu=name` option.

Deprecated options are not listed.

See also

- `--fpu=name`.

2.1.55 `--fpu=name`

This option enables you to determine the target FPU architecture.

If you specify this option, it overrides any implicit FPU option that appears on the command line, for example, where you use the `--cpu` option.

To obtain a full list of FPU architectures use the `--fpu=list` option.

Syntax

`--fpu=name`

Where *name* is one of:

none Selects no floating-point option. No floating-point code is to be used. This produces an error if your code contains **float** types.

vfpv3 Selects hardware vector floating-point unit conforming to architecture VFPv3.

vfpv2 Selects hardware vector floating-point unit conforming to architecture VFPv2. This is the default if you specify `--fpu=vfp`.

———— **Note** —————

If you enter `armcc --thumb --fpu=vfpv2` on the command line, the compiler compiles as much of the code using the Thumb instruction set as possible, but hard floating-point sensitive functions are compiled to ARM code. In this case, the value of the predefine `__thumb` is not correct.

If you specify either `vfp` or `vfpv2` with the `--arm` option for ARM C code you must use the `__softfp` keyword to ensure that your interworking ARM code is compiled to use software floating-point linkage.

softvfp Selects software floating-point linkage. This is the default if you do not specify a `--fpu` option, or if you select a CPU that does not have an FPU.

In previous releases of RVCT, if you specified `--fpu=softvfp` and a CPU with implicit VFP hardware, the linker chose a library that implemented the software floating-point calls using VFP instructions. This is no longer the case. If you require this legacy behavior, use `--fpu=softvfp+vfp`.

softvfp+vfpv2

Selects a floating-point library with software floating-point linkage that uses VFP instructions. Select this option if you are interworking Thumb code with ARM code on a system that implements a VFP unit.

If you select this option:

- Compiling with `--thumb` behaves in a similar way to `--fpu=softvfp` except that it links with floating-point libraries that use VFP instructions.
- Compiling with `--arm` behaves in a similar way to `--fpu=vfpv2` except that all functions are given software floating-point linkage. This means that functions pass and return floating-point arguments and results in the same way as `--fpu=softvfp`, but use VFP instructions internally.

Note

If you specify `softvfp+vfpv2` with the `--arm` or `--thumb` option for C code, it ensures that your interworking floating-point code is compiled to use software floating-point linkage.

softvfp+vfpv3

Selects a floating-point library with software floating-point linkage that uses VFPv3 instructions. Select this option if you are interworking Thumb code with ARM code on a system that implements a VFPv3 unit.

See also

- `--arm` on page 2-7
- `--cpu=name` on page 2-15
- `--thumb` on page 2-86
- `__softfp` on page 4-15
- *The `--fpu` option* on page 4-32 in the *Compiler User Guide*
- *Using the `--fpu` option* on page 4-34 in the *Compiler User Guide*.

2.1.56 `--[no_]friend_injection`

This option controls the visibility of friend declarations in C++.

In C++, it controls whether or not the name of a class or function that is declared only in friend declarations is visible when using the normal lookup mechanisms.

When friend names are declared, they are visible to these lookups. When friend names are not declared as required by the standard, function names are visible only when using argument-dependent lookup, and class names are never visible.

Note

The option `--friend_injection` is provided only as a migration aid for legacy source code that does not conform to the C++ standard. Its use is not recommended.

Mode

This option is effective only if the source language is C++.

Default

The default is `--no_friend_injection`.

See also

- *friend* on page 3-15.

2.1.57 `--gnu`

This option enables the GNU compiler extensions supported by the ARM compiler. The version of GCC the extensions are compatible with can be determined by inspecting the predefined macros `__GNUC__` and `__GNUC_MINOR__`.

See also

- `--c90` on page 2-13
- `--c99` on page 2-13
- `--cpp` on page 2-14
- `--[no_]strict` on page 2-84
- *GNU language extensions* on page 3-25
- *Compiler predefines* on page 4-103.

2.1.58 `--[no_]guiding_decls`

This option enables or disables the recognition of guiding declarations for template functions in C++.

A *guiding declaration* is a function declaration that matches an instance of a function template but has no explicit definition because its definition derives from the function template.

If `--no_guiding_decls` is combined with `--old_specializations`, a specialization of a non-member template function is not recognized. It is treated as a definition of an independent function.

Note

The option `--guiding_decls` is provided only as a migration aid for legacy source code that does not conform to the C++ standard. Its use is not recommended.

Mode

This option is effective only if the source language is C++.

Default

The default is `--no_guiding_decls`.

Example

```
template <class T> void f(T)
{
    ...
}
void f(int);
```

When regarded as a guiding declaration, `f(int)` is an instance of the template. Otherwise, it is an independent function so you must supply a definition.

See also

- `--apcs=qualifer...qualifier` on page 2-4
- `--[no_]old_specializations` on page 2-68.

2.1.59 --help

This option displays a summary of the main command-line options.

This is the default if you do not specify any options or source files.

See also

- `--show_cmdline` on page 2-81.
- `--vsn` on page 2-94

2.1.60 --[no_]hide_all

This option enables you to control symbol visibility when building SVr4 shared objects.

Usage

Use `--no_hide_all` to mark all extern definitions as `__declspec(dllexport)`, and to import all undefined references.

Default

The default is `--hide_all`.

See also

- `__declspec(dllexport)` on page 4-24
- `--symver_script` in *Controlling image contents* on page 2-16 in the *Linker Guide*.

2.1.61 -I`dir`[,`dir`,...]

This option adds the specified directory, or comma-separated list of directories, to the list of places that are searched to find included files.

If you specify more than one directory, the directories are searched in the same order as the `-I` options specifying them.

Syntax

`-Idir[,dir,...]`

Where:

`dir[,dir,...]` is a comma-separated list of directories to be searched for included files.

At least one directory must be specified.

When specifying multiple directories, do not include spaces between commas and directory names in the list.

See also

- `-Idir[,dir,...]` on page 2-51
- `--kandr_include` on page 2-52
- `--preinclude=filename` on page 2-75
- `--sys_include` on page 2-86

- *Include files* on page 2-13 in the *Compiler User Guide*.

2.1.62 --[no_]implicit_include

This option controls the implicit inclusion of source files as a method of finding definitions of template entities to be instantiated in C++.

Mode

This option is effective only if the source language is C++.

Default

The default is `--implicit_include`.

See also

- `--[no_]implicit_include_searches`
- *Implicit inclusion* on page 5-15.

2.1.63 --[no_]implicit_include_searches

This option controls how the compiler searches for implicit include files for templates in C++.

When the option `--implicit_include_searches` is selected, the compiler uses the search path to look for implicit include files based on partial names of the form *filename.**. The search path is determined by `-I`, `-J`, and the `RVCT31INC` environment variable.

When the option `--no_implicit_include_searches` is selected, the compiler looks for implicit include files based on the full names of files, including path names.

Mode

This option is effective only if the source language is C++.

Default

The default is `--no_implicit_include_searches`.

See also

- `-I`*dir*,*dir*,... on page 2-46
- `--[no_]implicit_include`

- *-Jdir[,dir,...]* on page 2-51
- *Implicit inclusion* on page 5-15
- *The search path* on page 2-14 in the *Compiler User Guide*.

2.1.64 `--[no_]implicit_typename`

This option controls the implicit determination, from context, whether a template parameter dependent name is a type or nontype in C++.

Note

The option `--implicit_typename` is provided only as a migration aid for legacy source code that does not conform to the C++ standard. Its use is not recommended.

Mode

This option is effective only if the source language is C++.

Default

The default is `--no_implicit_typename`.

Note

The `--implicit_typename` option has no effect unless you also specify `--no_parse_templates`.

See also

- *--[no_]dep_name* on page 2-22
- *--[no_]parse_templates* on page 2-70
- *Template instantiation* on page 5-15.

2.1.65 `--info=totals`

This option instructs the compiler to give totals of the object code and data size for each object file.

The compiler returns the same totals as `fromelf` returns when `fromelf -z` is used, in a similar format. The totals include embedded assembler sizes when embedded assembly exists in the source code.

Example

Code (inc. data)	RO Data	RW Data	ZI Data	Debug	File Name
3308	1556	0	44	10200	8402 dhry_1.o
Code (inc. data)	RO Data	RW Data	ZI Data	Debug	File Name
416	28	0	0	0	7722 dhry_2.o

The (inc. data) column gives the size of constants, string literals, and other data items used as part of the code. The Code column, shown in the example, *includes* this value.

See also

- *--list* on page 2-55
- *Code metrics* on page 4-11 in the *Compiler User Guide*
- *fromelf command syntax* on page 8-3 in the *Linker Guide*.

2.1.66 --[no_]inline

This option enables or disables the inlining of functions. Disabling the inlining of functions can help to improve the debug illusion.

When the option `--inline` is selected, the compiler considers inlining each function. Compiling your code with `--inline` does not guarantee that all functions are inlined. See *When is it practical for the compiler to inline?* on page 4-19 in the *Compiler User Guide* for more information about how the compiler decides to inline functions.

When the option `--no_inline` is selected, the compiler does not attempt to inline functions, other than functions qualified with `__forceinline`.

Default

The default is `--inline`.

See also

- *--[no_]autoinline* on page 2-9
- *--forceinline* on page 2-39
- *-Onum* on page 2-66
- *-Ospace* on page 2-68
- *-Otime* on page 2-69
- *__forceinline* on page 4-6
- *__inline* on page 4-9

- *Using linker feedback* on page 2-25 in the *Compiler User Guide*
- *Inlining* on page 4-19 in the *Compiler User Guide*.

2.1.67 --interleave

This option interleaves C or C++ source code line by line as comments within an assembly listing generated using the --asm option or -S option.

Usage

The action of --interleave depends on the combination of options used:

Table 2-3 Compiling with the ---interleave option

Compiler option	Action
--asm --interleave	Writes a listing to a file of the disassembly of the compiled source, interleaving the source code with the disassembly. The link step is also performed, unless the -c option is used. The disassembly is written to a text file whose name defaults to the name of the input file with the filename extension .txt
-S --interleave	Writes a listing to a file of the disassembly of the compiled source, interleaving the source code with the disassembly. The disassembly is written to a text file whose name defaults to the name of the input file with the filename extension .txt

Restrictions

- You cannot reassemble an assembly listing generated with --asm --interleave or -S --interleave.
- Preprocessed source files contain #line directives. When compiling preprocessed files using --asm --interleave or -S --interleave, the compiler searches for the original files indicated by any #line directives, and uses the correct lines from those files. This ensures that compiling a preprocessed file gives exactly the same output and behavior as if the original files were compiled.

If the compiler cannot find the original files, it is unable to interleave the source. Therefore, if you have preprocessed source files with #line directives, but the original unpreprocessed files are not present, you must remove all the #line directives before you compile with --interleave.

See also

- `--asm` on page 2-8
- `-S` on page 2-80.

2.1.68 `-Jdir[,dir,...]`

This option adds the specified directory, or comma-separated list of directories, to the list of system includes.

The RVCT31INC environment variable is set to the default system include path unless you use `-J` to override it. Angle-bracketed include files are searched for first in the list of system includes, followed by any include list specified with the option `-I`.

Note

On Windows systems, you must enclose RVCT31INC in double quotes if you specify this environment variable on the command line, because the default path defined by the variable contains spaces. For example:

```
armcc -J"%RVCT31INC%" -c main.c
```

Syntax

`-Jdir[,dir,...]`

Where:

`dir[,dir,...]` is a comma-separated list of directories to be added to the list of system includes.

At least one directory must be specified.

When specifying multiple directories, do not include spaces between commas and directory names in the list.

See also

- `-Idir[,dir,...]` on page 2-46
- `--kandr_include` on page 2-52
- `--preinclude=filename` on page 2-75
- `--sys_include` on page 2-86
- *Include files* on page 2-13 in the *Compiler User Guide*.

2.1.69 --kandr_include

This option ensures that Kernighan and Ritchie search rules are used for locating included files.

The current place is defined by the original source file and is not stacked. If you do not use this option, Berkeley-style searching is used.

See also

- *-Idir[,dir,...]* on page 2-46
- *-Jdir[,dir,...]* on page 2-51
- *--preinclude=filename* on page 2-75
- *--sys_include* on page 2-86
- *Include files* on page 2-13 in the *Compiler User Guide*.
- *The current place* on page 2-13 in the *Compiler User Guide*.

2.1.70 -Lopt

This option specifies command-line options to pass to the linker when a link step is being performed after compilation.

Options can be passed when creating a partially-linked object or an executable image.

Syntax

`-Lopt`

Where:

opt is a command-line option to pass to the linker.

Restrictions

If an unsupported Linker option is passed to it using `-L`, an error is generated.

Example

```
armcc main.c -L--map
```

See also

- *-Aopt* on page 2-2
- *--show_cmdline* on page 2-81.

2.1.71 --library_interface=lib

This option enables the generation of code that is compatible with the selected library type.

Syntax

```
--library_interface=lib
```

Where *lib* is one of:

rvct	Specifies that the compiler output works with the RVCT runtime libraries.
aeabi_clib90	Specifies that the compiler output works with any ISO C90 library compliant with the <i>ARM Embedded Application Binary Interface</i> (AEABI).
aeabi_clib99	Specifies that the compiler output works with any ISO C99 library compliant with the <i>ARM Embedded Application Binary Interface</i> (AEABI).
aeabi_clib	Specifies that the compiler output works with any ISO C library compliant with the <i>ARM Embedded Application Binary Interface</i> (AEABI). Selecting the option <code>--library_interface=aeabi_clib</code> is equivalent to specifying either <code>--library_interface=aeabi_clib90</code> or <code>--library_interface=aeabi_clib99</code> , depending on the choice of source language used. The choice of source language is dependent both on the command-line options selected and on the filename suffixes used.
aeabi_glibc	Specifies that the compiler output works with an AEABI-compliant version of the GNU C library.

Default

If you do not specify `--library_interface`, the compiler assumes `--library_interface=rvct`.

Usage

- Use the option `--library_interface=rvct` to exploit the full range of compiler and library optimizations when linking.

- Use an option of the form `--library_interface=aeabi_*` when linking with an ABI-compliant C library. Options of the form `--library_interface=aeabi_*` ensure that the compiler does not generate calls to any optimized functions provided by the RVCT C library.

Example

When your code calls functions provided by an embedded operating system that replace functions provided by the RVCT C library, compile your code with `--library_interface=aeabi_clib` to disable calls to any special RVCT variants of the library functions replaced by the operating system.

See also

- *ABI for the ARM Architecture compliance* on page 1-3 in the *Libraries Guide*.

2.1.72 `--library_type=lib`

This option enables the selected library to be used at link time.

———— Note —————

Use this option with the linker to override all other `--library_type` options.

Syntax

`--library_type=lib`

Where *lib* is one of:

- | | |
|--------------------------|--|
| <code>standardlib</code> | Specifies that the full RVCT runtime libraries are selected at link time.
Use this option to exploit the full range of compiler optimizations when linking. |
| <code>microlib</code> | Specifies that the C micro-library (microlib) is selected at link time. |

Default

If you do not specify `--library_type`, the compiler assumes `--library_type=standardlib`.

See also

- *Building an application with microlib* on page 3-4 in the *Libraries Guide*

- `--library_type=lib` in *Specifying an input file list* on page 2-6 in the *Linker Guide*.

2.1.73 `--list`

This option instructs the compiler to generate raw listing information for a source file. The name of the raw listing file defaults to the name of the input file with the filename extension `.lst`.

If you specify multiple source files on the command line then raw listing information is generated for only the first of the specified files.

Usage

Typically, raw listing information is used to generate a formatted listing. The raw listing file contains raw source lines, information on transitions into and out of include files, and diagnostics generated by the compiler. Each line of the listing file begins with any of the following key characters that identifies the type of line:

- | | |
|---|--|
| N | A normal line of source. The rest of the line is the text of the line of source. |
| X | The expanded form of a normal line of source. The rest of the line is the text of the line. This line appears following the N line, and only if the line contains nontrivial modifications. Comments are considered trivial modifications, and macro expansions, line splices, and trigraphs are considered nontrivial modifications. Comments are replaced by a single space in the expanded-form line. |
| S | A line of source skipped by an <code>#if</code> or similar. The rest of the line is text. |

————— **Note** —————

The `#else`, `#elseif`, or `#endif` that ends a skip is marked with an N.

- | | |
|---|--|
| L | Indicates a change in source position. That is, the line has a format similar to the <code>#</code> line-identifying directive output by the preprocessor: |
|---|--|

L *line-number* "*filename*" *key*

where *key* can be:

- | | |
|---|---------------------------------|
| 1 | For entry into an include file. |
| 2 | For exit from an include file. |

Otherwise, *key* is omitted. The first line in the raw listing file is always an L line identifying the primary input file. L lines are also output for #line directives where *key* is omitted. L lines indicate the source position of the following source line in the raw listing file.

R/W/E Indicates a diagnostic, where:

R Indicates a remark.

W Indicates a warning.

E Indicates an error.

The line has the form:

```
type "filename" line-number column-number message-text
```

where *type* can be R, W, or E.

Errors at the end of file indicate the last line of the primary source file and a column number of zero.

Command-line errors are errors with a filename of "<command line>". No line or column number is displayed as part of the error message.

Internal errors are errors with position information as usual, and message-text beginning with (Internal fault).

When a diagnostic message displays a list, for example, all the contending routines when there is ambiguity on an overloaded call, the initial diagnostic line is followed by one or more lines with the same overall format. However, the code letter is the lowercase version of the code letter in the initial line. The source position in these lines is the same as that in the corresponding initial line.

Example

```
/* main.c */
#include <stdbool.h>

int main(void)
{
    return(true);
}
```

Compiling this code with the option `--list` produces the raw listing file:

```
L 1 "main.c"
N#include <stdbool.h>
L 1 "...\\include\\...\\stdbool.h" 1
N/* stdbool.h */
N
...
```

```

N #ifndef __cplusplus /* In C++, 'bool', 'true' and 'false' and keywords */
N #define bool _Bool
N #define true 1
N #define false 0
N #endif
...
L 2 "main.c" 2
N
Nint main(void)
N{
N    return(true);
X    return(1);
N}

```

See also

- `--asm` on page 2-8
- `-c` on page 2-12
- `--depend=filename` on page 2-23
- `--depend_format=string` on page 2-24
- `--info=totals` on page 2-48
- `--interleave` on page 2-50
- `--md` on page 2-60
- `-S` on page 2-80
- *Severity of diagnostic messages* on page 5-3 in the *Compiler User Guide*.

2.1.74 `--littleend`

This option instructs to the compiler to generate code for an ARM processor using little-endian memory.

With little-endian memory, the least significant byte of a word has the lowest address.

Default

The compiler assumes `--littleend` unless `--bigend` is explicitly specified.

See also

- `--bigend` on page 2-10.

2.1.75 `--locale=lang_country`

This option switches the default locale for source files to the one you specify in *lang_country*.

Syntax

`--locale=lang_country`

Where:

lang_country is the new default locale.

Use this option in combination with `--multibyte_chars`.

Restrictions

The locale name might be case-sensitive, depending on the host platform.

The permitted settings of locale are determined by the host platform.

Ensure that you have installed the appropriate locale support for the host platform.

Example

To compile Japanese source files on an English-based Windows workstation, use:

```
--multibyte_chars --locale=japanese
```

and on a UNIX workstation use:

```
--multibyte_chars --locale=ja_JP
```

See also

- `--message_locale=lang_country[.codepage]` on page 2-61
- `--[no_]multibyte_chars` on page 2-62.

2.1.76 `--loose_implicit_cast`

This option makes illegal implicit casts legal, such as implicit casts of a non-zero integer to a pointer.

Example

```
int *p = 0x8000;
```

Compiling this example without the option `--loose_implicit_cast`, generates an error.

Compiling this example with the option `--loose_implicit_cast`, generates a warning message, that you can suppress.

2.1.77 `--[no_]lower_ropi`

This option enables or disables less restrictive C when compiling with `--apcs=/ropi`.

Default

The default is `--no_lower_ropi`.

Note

If you compile with `--lower_ropi`, then the static initialization is done at runtime by the C++ constructor mechanism for both C and C++ code. This enables these static initializations to work with ROPI code.

See also

- `--apcs=qualifer...qualifier` on page 2-4
- `--[no_]lower_rwpi`
- *Position independence qualifiers* on page 2-23 in the *Compiler User Guide*.

2.1.78 `--[no_]lower_rwpi`

This option enables or disables less restrictive C and C++ when compiling with `--apcs=/rwpi`.

Default

The default is `--lower_rwpi`.

Note

If you compile with `--lower_rwpi`, then the static initialization is done at runtime by the C++ constructor mechanism, even for C. This enables these static initializations to work with RWPI code.

See also

- `--apcs=qualifer...qualifier` on page 2-4
- `--[no_]lower_ropi`
- *Position independence qualifiers* on page 2-23 in the *Compiler User Guide*.

2.1.79 -M

This option instructs the compiler to produce a list of make file dependency lines suitable for use by a make utility.

The compiler executes only the preprocessor step of the compilation. By default, output is on the standard output stream.

If you specify multiple source files, a single dependency file is created.

If you specify the `-o filename` option, the dependency lines generated on standard output make reference to `filename.o`, and not to `source.o`. However, no object file is produced with the combination of `-M -o filename`.

Use the `--md` option to generate dependency lines and object files for each source file.

Example

You can redirect output to a file by using standard UNIX and MS-DOS notation, for example:

```
armcc -M source.c > Makefile
```

See also

- `-C` on page 2-13
- `--[no_]depend_system_headers` on page 2-25
- `-E` on page 2-33
- `--md`
- `-o filename` on page 2-65.

2.1.80 --md

This option instructs the compiler to compile the source and write make file dependency lines to a file.

The output file is suitable for use by a make utility.

The compiler names the file `filename.d`, where `filename` is the name of the source file. If you specify multiple source files, a dependency file is created for each source file.

See also

- `--depend=filename` on page 2-23
- `--depend_format=string` on page 2-24
- `--[no_]depend_system_headers` on page 2-25

- `-M` on page 2-60
- `-o filename` on page 2-65.

2.1.81 `--message_locale=lang_country[.codepage]`

This option switches the default language for the display of error and warning messages to the one you specify in `lang_country` or `lang_country.codepage`.

Syntax

`--message_locale=lang_country[.codepage]`

Where:

`lang_country[.codepage]`

is the new default language for the display of error and warning messages.

The permitted languages are independent of the host platform.

The following settings are supported:

- `en_US`
- `zh_CN`
- `ko_KR`
- `ja_JP`.

Default

If you do not specify `--message_locale`, the compiler assumes `--message_locale=en_US`.

Restrictions

Ensure that you have installed the appropriate locale support for the host platform.

The locale name might be case-sensitive, depending on the host platform.

The ability to specify a codepage, and its meaning, depends on the host platform.

Errors

If you specify a setting that is not supported, the compiler generates an error message.

Example

To display messages in Japanese, use:

```
--message_locale=ja_JP
```

See also

- `--locale=lang_country` on page 2-57
- `--[no_]multibyte_chars`.

2.1.82 `--min_array_alignment=opt`

This option enables you to specify the minimum alignment of arrays.

Syntax

```
--min_array_alignment=opt
```

Where:

<i>opt</i>	specifies the minimum alignment of arrays. The value of <i>opt</i> is:
1	byte alignment, or unaligned
2	two-byte, halfword alignment
4	four-byte, word alignment
8	eight-byte, doubleword alignment.

Default

If you do not specify a `--min_array_alignment` option, the compiler assumes `--min_array_alignment=1`.

Example

Compiling the following code with `--min_array_alignment=8` gives the alignment described in the comments:

```
char arr_c1[1];    // alignment == 8
char c1;           // alignment == 1
```

See also

- `__align` on page 4-2
- `__ALIGNOF__` on page 4-4.

2.1.83 `--[no_]multibyte_chars`

This option enables or disables processing for multibyte character sequences in comments, string literals, and character constants.

Default

The default is `--no_multibyte_chars`.

Usage

Multibyte encodings are used for character sets such as the Japanese *Shift-Japanese Industrial Standard* (Shift-JIS).

See also

- `--locale=lang_country` on page 2-57
- `--message_locale=lang_country[.codepage]` on page 2-61.

2.1.84 --[no_]multifile

This option enables or disables multifile compilation.

When `--multifile` is selected, the compiler performs optimizations across all files specified on the command line, instead of on each individual file. The specified files are compiled into one single object file.

The combined object file is named after the first source file you specify on the command line. To specify a different name for the combined object file, use the `-o filename` option.

An empty object file is created for each subsequent source file specified on the command line to meet the requirements of standard make systems.

————— Note —————

Compiling with `--multifile` has no effect if only a single source file is specified on the command line.

Default

The default is `--no_multifile`, unless the option `-O3` is specified.

If the option `-O3` is specified, then the default is `--multifile`.

Usage

When `--multifile` is selected, the compiler might be able to perform additional optimizations by compiling across several source files.

There is no limit to the number of source files that can be specified on the command line, but ten files is a practical limit, because `--multifile` requires large amounts of memory while compiling. For the best optimization results, choose small groups of functionally related source files.

Example

```
armcc -c --multifile test1.c ... testn.c -o test.o
```

The resulting object file is named `test.o`, instead of `test1.c`, and empty object files `test2.o` to `testn.o` are created for each source file `test1.c ... testn.c` specified on the command line.

See also

- `-c` on page 2-12
- `--default_extension=ext` on page 2-22
- `-o filename` on page 2-65
- `-Onum` on page 2-66.

2.1.85 `--[no_]nonstd_qualifier_deduction`

This option controls whether or not nonstandard template argument deduction is to be performed in the qualifier portion of a qualified name in C++.

With this feature enabled, a template argument for the template parameter `T` can be deduced in contexts like `A<T>::B` or `T::B`. The standard deduction mechanism treats these as non deduced contexts that use the values of template parameters that were either explicitly specified or deduced elsewhere.

———— Note —————

The option `--nonstd_qualifier_deduction` is provided only as a migration aid for legacy source code that does not conform to the C++ standard. Its use is not recommended.

Mode

This option is effective only if the source language is C++.

Default

The default is `--no_nonstd_qualifier_deduction`.

2.1.86 *-o filename*

This option specifies the name of the output file. The full name of the output file produced depends on the combination of options used, as described in Table 2-4 and Table 2-5.

Syntax

If you specify a *-o* option, the compiler names the output file according to the conventions of Table 2-4.

Table 2-4 Compiling with the *-o* option

Compiler option	Action	Usage notes
<i>-o-</i>	writes output to the standard output stream	<i>filename</i> is <i>-</i> . <i>-S</i> is assumed unless <i>-E</i> is specified.
<i>-o filename</i>	produces an executable image with name <i>filename</i>	
<i>-c -o filename</i>	produces an object file with name <i>filename</i>	
<i>-S -o filename</i>	produces an assembly language file with name <i>filename</i>	
<i>-E -o filename</i>	produces a file containing preprocessor output with name <i>filename</i>	

If you do not specify a *-o* option, the compiler names the output file according to the conventions of Table 2-5.

Table 2-5 Compiling without the *-o* option

Compiler option	Action	Usage notes
<i>-c</i>	produces an object file whose name defaults to the name of the input file with the filename extension <i>.o</i>	

Table 2-5 Compiling without the -o option (continued)

Compiler option	Action	Usage notes
-S	produces an output file whose name defaults to the name of the input file with the filename extension .s	
-E	writes output from the preprocessor to the standard output stream	
(No option)	produces an executable image with the default name of __image.axf	none of -o, -c, -E or -S is specified on the command line

Note
This option overrides the `--default_extension` option.

See also

- `--asm` on page 2-8
- `-c` on page 2-12
- `--default_extension=ext` on page 2-22
- `--depend=filename` on page 2-23
- `--depend_format=string` on page 2-24
- `-E` on page 2-33
- `--interleave` on page 2-50
- `--list` on page 2-55
- `--md` on page 2-60
- `-S` on page 2-80.

2.1.87 `-Onum`

This option specifies the level of optimization to be used when compiling source files.

Syntax

`-Onum`

Where *num* is one of the following:

- 0 Minimum optimization. Turns off most optimizations. It gives the best possible debug view and the lowest level of optimization.

- 1 Restricted optimization. Removes unused inline functions and unused static functions. Turns off optimizations that seriously degrade the debug view. If used with `--debug`, this option gives a satisfactory debug view with good code density.
- 2 High optimization. If used with `--debug`, the debug view might be less satisfactory because the mapping of object code to source code is not always clear.
This is the default optimization level.
- 3 Maximum optimization. `-O3` performs the same optimizations as `-O2` however the balance between space and time optimizations in the generated code is more heavily weighted towards space or time compared with `-O2`. That is:
 - `-O3 -Otime` aims to produce faster code than `-O2 -Otime`, at the risk of increasing your image size
 - `-O3 -Ospace` aims to produce smaller code than `-O2 -Ospace`, but performance might be degraded.

In addition, `-O3` performs extra optimizations that are more aggressive, such as:

 - High-level scalar optimizations, including loop unrolling, for `-O3 -Otime`. This can give significant performance benefits at a small code size cost, but at the risk of a longer build time.
 - More aggressive inlining and automatic inlining for `-O3 -Otime`.
 - Multifile compilation by default.

Note

The performance of floating-point code can be influenced by selecting an appropriate numerical model using the `--fpmodel` option.

Note

Do not rely on the implementation details of these optimizations, because they might change in future releases.

Default

If you do not specify `-Onum`, the compiler assumes `-O2`.

See also

- `--[no_]autoinline` on page 2-9
- `--[no_]debug` on page 2-21
- `--forceinline` on page 2-39
- `--fpmode=model` on page 2-39
- `--[no_]inline` on page 2-49
- `--[no_]multifile` on page 2-63
- `-Ospace`
- `-Otime` on page 2-69
- *Optimizing code* on page 4-2 in the *Compiler User Guide*.

2.1.88 `--[no_]old_specializations`

This option controls the acceptance of old-style template specializations in C++.

Old-style template specializations do not use the `template<>` syntax.

———— Note ————

The option `--old_specializations` is provided only as a migration aid for legacy source code that does not conform to the C++ standard. Its use is not recommended.

Mode

This option is effective only if the source language is C++.

Default

The default is `--no_old_specializations`.

2.1.89 `-Ospace`

This option instructs the compiler to perform optimizations to reduce image size at the expense of a possible increase in execution time.

Use this option if code size is more critical than performance. For example, when the `-Ospace` option is selected, large structure copies are done by out-of-line function calls instead of inline code.

If required, you can compile the time-critical parts of your code with `-Otime`, and the rest with `-Ospace`.

Default

If you do not specify either `-Ospace` or `-Otime`, the compiler assumes `-Ospace`.

See also

- `-Otime`
- `-Onum` on page 2-66
- `#pragma Onum` on page 4-57
- `#pragma Ospace` on page 4-58
- `#pragma Otime` on page 4-58.

2.1.90 -Otime

This option instructs the compiler to perform optimizations to reduce execution time at the expense of a possible increase in image size.

Use this option if execution time is more critical than code size. If required, you can compile the time-critical parts of your code with `-Otime`, and the rest with `-Ospace`.

Default

If you do not specify `-Otime`, the compiler assumes `-Ospace`.

Example

When the `-Otime` option is selected, the compiler compiles:

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

as:

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

See also

- `--[no_]multifile` on page 2-63
- `-Onum` on page 2-66
- `-Ospace` on page 2-68
- `#pragma Onum` on page 4-57
- `#pragma Ospace` on page 4-58

- *#pragma Otime* on page 4-58.

2.1.91 --[no_]parse_templates

This option enables or disables the parsing of non class templates in their generic form in C++, that is, when the template is defined and before it is instantiated.

Note

The option `--no_parse_templates` is provided only as a migration aid for legacy source code that does not conform to the C++ standard. Its use is not recommended.

Mode

This option is effective only if the source language is C++.

Default

The default is `--parse_templates`.

Note

`--no_parse_templates` cannot be used with `--dep_name`, because parsing is done by default if dependent name processing is enabled. Combining these options generates an error.

See also

- *--[no_]dep_name* on page 2-22
- *Template instantiation* on page 5-15.

2.1.92 --pch

This option instructs the compiler to use a PCH file if it exists, and to create a PCH file otherwise.

When the option `--pch` is specified, the compiler searches for a PCH file with the name *filename.pch*, where *filename.** is the name of the primary source file. The compiler uses the PCH file *filename.pch* if it exists, and creates a PCH file named *filename.pch* in the same directory as the primary source file otherwise.

Restrictions

This option has no effect if you include either the option `--use_pch=filename` or the option `--create_pch=filename` on the same command line.

See also

- `--create_pch=filename` on page 2-18
- `--pch_dir=dir`
- `--[no_]pch_messages` on page 2-72
- `--[no_]pch_verbose` on page 2-72
- `--use_pch=filename` on page 2-89
- `#pragma hdrstop` on page 4-55
- `#pragma no_pch` on page 4-57
- *Precompiled header files* on page 2-16 in the *Compiler User Guide*.

2.1.93 `--pch_dir=dir`

This option enables you to specify the directory where PCH files are stored. The directory is accessed whenever PCH files are created or used.

You can use this option with automatic or manual PCH mode.

Syntax

`--pch_dir=dir`

Where:

dir is the name of the directory where PCH files are stored.

Errors

If the specified directory *dir* does not exist, the compiler generates an error.

See also

- `--create_pch=filename` on page 2-18
- `--pch` on page 2-70
- `--[no_]pch_messages` on page 2-72
- `--[no_]pch_verbose` on page 2-72
- `--use_pch=filename` on page 2-89
- `#pragma hdrstop` on page 4-55
- `#pragma no_pch` on page 4-57

- *Precompiled header files* on page 2-16 in the *Compiler User Guide*.

2.1.94 --[no_]pch_messages

This option enables or disables the display of messages indicating that a PCH file is used in the current compilation.

Default

The default is `--pch_messages`.

See also

- `--create_pch=filename` on page 2-18
- `--pch` on page 2-70
- `--pch_dir=dir` on page 2-71
- `--[no_]pch_verbose`
- `--use_pch=filename` on page 2-89
- `#pragma hdrstop` on page 4-55
- `#pragma no_pch` on page 4-57
- *Precompiled header files* on page 2-16 in the *Compiler User Guide*.

2.1.95 --[no_]pch_verbose

This option enables or disables the display of messages giving reasons why a file cannot be precompiled.

In automatic PCH mode, this option ensures that for each PCH file that cannot be used for the current compilation, a message is displayed giving the reason why the file cannot be used.

Default

The default is `--no_pch_verbose`.

See also

- `--create_pch=filename` on page 2-18
- `--pch` on page 2-70
- `--pch_dir=dir` on page 2-71
- `--[no_]pch_messages`
- `--use_pch=filename` on page 2-89
- `#pragma hdrstop` on page 4-55

- *#pragma no_pch* on page 4-57
- *Precompiled header files* on page 2-16 in the *Compiler User Guide*.

2.1.96 --pending_instantiations=*n*

This option specifies the maximum number of concurrent instantiations of a template in C++.

Syntax

--pending_instantiations=*n*

Where:

n is the maximum number of concurrent instantiations permitted.
If *n* is zero, there is no limit.

Mode

This option is effective only if the source language is C++.

Default

If you do not specify a --pending_instantiations option, then the compiler assumes --pending_instantiations=64.

Usage

Use this option to detect runaway recursive instantiations.

2.1.97 --pointer_alignment=*num*

This option specifies the unaligned pointer support required for an application.

Syntax

--pointer_alignment=*num*

Where *num* is one of:

- | | |
|---|--|
| 1 | Treats accesses through pointers as having an alignment of one, that is, byte-aligned or unaligned. |
| 2 | Treats accesses through pointers as having an alignment of at most two, that is, at most halfword aligned. |

- | | |
|---|---|
| 4 | Treats accesses through pointers as having an alignment of at most four, that is, at most word aligned. |
| 8 | Accesses through pointers have normal alignment, that is, at most doubleword aligned. |

Usage

This option can help you port source code that has been written for architectures without alignment requirements. You can achieve finer control of access to unaligned data, with less impact on the quality of generated code, using the `__packed` qualifier.

Restrictions

De-aligning pointers might increase the code size, even on CPUs with unaligned access support. This is because only a subset of the load and store instructions benefit from unaligned access support. The compiler is unable to use multiple-word transfers or coprocessor-memory transfers, including hardware floating-point loads and stores, directly on unaligned memory objects.

———— Note ————

- Code size might increase significantly when compiling for CPUs without hardware support for unaligned access, for example, pre-v6 architectures.
- This option does not affect the placement of objects in memory, nor the layout and padding of structures.

See also

- `__packed` on page 4-11
- *Aligning data* on page 4-24 in the *Compiler User Guide*.

2.1.98 --[no_]project=*filename*

The option `--project=filename` instructs the compiler to load the project template file specified by *filename*.

———— Note ————

To use *filename* as a default project file, set the RVDS_PROJECT environment variable to *filename*.

The option `--no_project` prevents the default project template file specified by the environment variable RVDS_PROJECT from being used.

Syntax

`--project=filename`

`--no_project`

Where:

filename is the name of a project template file.

Restrictions

Options from a project template file are only set when they do not conflict with options already set on the command line. If an option from a project template file conflicts with an existing command-line option, the command-line option takes precedence.

Example

Consider the following project template file:

```
<!-- suiteconf.cfg -->
<suiteconf name="Platform Baseboard for ARM926EJ-S">
  <tool name="armcc">
    <cmdline>
      --cpu=ARM926EJ-S
      --fpu=vfpv2
    </cmdline>
  </tool>
</suiteconf>
```

When the `RVDS_PROJECT` environment variable is set to point to this file, the command:

```
armcc -c foo.c
```

results in an actual command line of:

```
armcc --cpu=ARM926EJ-S --fpu=VFPv2 -c foo.c
```

See also

- `--reinitialize_workdir` on page 2-77
- `--workdir=directory` on page 2-95.

2.1.99 `--preinclude=filename`

This option instructs the compiler to include the source code of the specified file at the beginning of the compilation.

Syntax

`--preinclude=filename`

Where:

`filename` is the name of the file whose source code is to be included.

Usage

This option can be used to establish standard macro definitions. The *filename* is searched for in the directories on the include search list.

See also

- `-Idir[dir,...]` on page 2-46
- `-Jdir[dir,...]` on page 2-51
- `--kandr_include` on page 2-52
- `--sys_include` on page 2-86
- *Include files* on page 2-13 in the *Compiler User Guide*.

2.1.100 `--[no_]reduce_paths`

This option enables or disables the elimination of redundant path name information in file paths.

When elimination of redundant path name information is enabled, the compiler removes sequences of the form `xyz\..` from directory paths passed to the operating system. This includes system paths constructed by the compiler itself, for example, for `#include` searching.

———— **Note** ————

The removal of sequences of the form `xyz\..` might not be valid if `xyz` is a link.

Mode

This option is effective on Windows systems only.

Usage

Windows systems impose a 260 character limit on file paths. Where path names exist whose absolute names expand to longer than 260 characters, you can use the `--reduce_paths` option to reduce absolute path name length by matching up directories with corresponding instances of `..` and eliminating the `directory/..` sequences in pairs.

Note

It is recommended that you avoid using long and deeply nested file paths, in preference to minimizing path lengths using the `--reduce_paths` option.

Default

The default is `--no_reduce_paths`.

Example

Compiling the file

```
..\..\..\xyzy\xyzy\objects\file.c
```

from the directory

```
\foo\bar\baz\gazonk\quux\bop
```

results in an actual path of

```
\foo\bar\baz\gazonk\quux\bop..\..\..\xyzy\xyzy\objects\file.o
```

Compiling the same file from the same directory using the option `--reduce_paths` results in an actual path of

```
\foo\bar\baz\xyzy\xyzy\objects\file.c
```

2.1.101 --reinitialize_workdir

This option enables you to reinitialize the project template working directory set using `--workdir`.

When the directory set using `--workdir` refers to an existing working directory containing modified project template files, specifying this option causes the working directory to be deleted and recreated with new copies of the original project template files.

Restrictions

This option must be used in combination with the `--workdir` option.

See also

- `--[no_]project=filename` on page 2-74
- `--workdir=directory` on page 2-95.

2.1.102 --remarks

This option instructs the compiler to issue remark messages, such as warning of padding in structures.

Note

The compiler does not issue remarks by default.

See also

- `--[no_]brief_diagnostics` on page 2-10
- `--diag_error=tag[,tag,...]` on page 2-26
- `--diag_remark=tag[,tag,...]` on page 2-27
- `--diag_style={arm|lde|gnu}` on page 2-27
- `--diag_suppress=tag[,tag,...]` on page 2-28
- `--diag_warning=tag[,tag,...]` on page 2-30
- `--errors=filename` on page 2-34
- `-W` on page 2-94
- `--[no_]wrap_diagnostics` on page 2-96.

2.1.103 --remove_unneeded_entities, --no_remove_unneeded_entities

These options control whether debug information is generated for all source symbols, or only for those symbols actually used.

Usage

Use `--remove_unneeded_entities` to reduce debug object and image file sizes. Faster linkage times can also be achieved.

Caution

Although `--remove_unneeded_entities` can help to reduce the amount of debug information generated per file, it has the disadvantage of reducing the number of debug sections that are common to many files. This reduces the number of common debug sections that the linker is able to remove at final link time, and can result in a final debug image that is larger than necessary. For this reason, use `--remove_unneeded_entities` only when necessary.

Restrictions

The effects of these options are restricted to debug information.

Default

Removal of unneeded entities is disabled by default.

See also

- The DWARF Debugging Standard at <http://dwarfstd.org/>.

2.1.104 `--[no_]restrict`

This option enables or disables the use of the C99 keyword **restrict**.

———— Note —————

The alternative keywords `__restrict` and `__restrict__` are supported as synonyms for **restrict**. These alternative keywords are always available, regardless of the use of the `--restrict` option.

Default

When compiling ISO C99 source code, use of the C99 keyword **restrict** is enabled by default.

When compiling ISO C90 or ISO C++ source code, use of the C99 keyword **restrict** is disabled by default.

See also

- *restrict* on page 3-8.

2.1.105 `--[no_]rtti`

This option controls support for the RTTI features `dynamic_cast` and `typeid` in C++.

Mode

This option is effective only if the source language is C++.

Default

The default is `--rtti`.

See also

- `--[no_]dllimport_runtime` on page 2-32.

2.1.106 -S

This option instructs the compiler to output the disassembly of the machine code generated by the compiler to a file.

Unlike the `--asm` option, object modules are not generated. The name of the assembly output file defaults to *filename.s* in the current directory, where *filename* is the name of the source file stripped of any leading directory names. The default filename can be overridden with the `-o` option.

You can use `armasm` to assemble the output file and produce object code. The compiler adds `ASSERT` directives for command-line options such as AAPCS variants and byte order to ensure that compatible compiler and assembler options are used when reassembling the output. You must specify the same AAPCS settings to both the assembler and the compiler.

See also

- `--asm` on page 2-8
- `-c` on page 2-12
- `--info=totals` on page 2-48
- `--interleave` on page 2-50
- `--list` on page 2-55
- `-o filename` on page 2-65.

2.1.107 --[un]signed_bitfields

This option makes bitfields of type `int` signed or unsigned.

The C Standard specifies that if the type specifier used in declaring a bitfield is either `int`, or a `typedef` name defined as `int`, then whether the bitfield is signed or unsigned is dependent on the implementation.

Default

The default is `--unsigned_bitfields`.

Note

The AAPCS requirement for bitfields to default to unsigned on ARM, is relaxed in version 2.03 of the standard.

Example

```
typedef int integer;

struct
{
    integer x : 1;
} bf;
```

Compiling this code with `--signed_bitfields` causes to be treated as a signed bitfield.

2.1.108 --[un]signed_chars

This option makes the **char** type signed or unsigned.

When **char** is signed, the macro `__FEATURE_SIGNED_CHAR` is defined by the compiler.

Note

Care must be taken when mixing translation units that have been compiled with and without this option, and that share interfaces or data structures.

The ARM ABI defines **char** as an unsigned byte, and this is the interpretation used by the C++ libraries supplied with RVCT.

Default

The default is `--unsigned_chars`.

See also

- *Predefined macros* on page 4-103.

2.1.109 --show_cmdline

This option shows how the command-line options are processed.

The commands are shown in their preferred form, and the contents of any via files are expanded.

See also

- *-Aopt* on page 2-2
- *-Lopt* on page 2-52
- *--via=filename* on page 2-92.

2.1.110 `--split_ldm`

This option instructs the compiler to split LDM and STM instructions into two or more LDM or STM instructions.

When `--split_ldm` is selected, the maximum number of register transfers for an LDM or STM instruction is limited to:

- five, for all STMs
- five, for LDMs that do not load the PC
- four, for LDMs that load the PC.

Where register transfers beyond these limits are required, multiple LDM or STM instructions are used.

Note

This option is deprecated and will be removed in the next release.

Usage

The `--split_ldm` option can be used to reduce interrupt latency on ARM systems that:

- do not have a cache or a write buffer, for example, a cacheless ARM7TDMI
- use zero-wait-state, 32-bit memory.

Note

Using `--split_ldm` increases code size and decreases performance slightly.

Restrictions

- Inline assembler LDM and STM instructions are split by default when `--split_ldm` is used. However, the compiler might subsequently recombine the separate instructions into an LDM or STM.
- VFP FLDm or FSTM instructions are not split when `--split_ldm` is used.

- Some target hardware does not benefit from code built with `--split_ldm`. For example:
 - It has no significant benefit for cached systems, or for processors with a write buffer.
 - It has no benefit for systems with non zero-wait-state memory, or for systems with slow peripheral devices. Interrupt latency in such systems is determined by the number of cycles required for the slowest memory or peripheral access. Typically, this is much greater than the latency introduced by multiple register transfers.

See also

- *Instruction expansion* on page 6-9 in the *Compiler User Guide*.

2.1.111 `--split_sections`

This option instructs the compiler to generate one ELF section for each function in the source file.

Output sections are named with the same name as the function that generates the section, but with an `i.` prefix.

————— Note —————

If you want to place specific data items or structures in separate sections, mark them individually with `__attribute__((section(...)))`.

If you want to remove unused functions, it is recommended that you use the linker feedback optimization in preference to this option. This is because linker feedback produces smaller code by avoiding the overhead of splitting all sections.

Restrictions

This option reduces the potential for sharing addresses, data, and string literals between functions. Consequently, it might increase code size slightly for some functions.

Example

```
int f(int x)
{
    return x+1;
}
```

Compiling this code with `--split_sections` produces:

```

        AREA ||i.f||, CODE, READONLY, ALIGN=2
f PROC
    ADD     r0,r0,#1
    BX      lr
    ENDP

```

See also

- `--[no_]data_reorder` on page 2-20
- `--feedback=filename` on page 2-37
- `--[no_]multifile` on page 2-63
- `__attribute__((section))` on page 4-34
- `#pragma arm section [section_sort_list]` on page 4-50
- *Using linker feedback* on page 2-25 in the *Compiler User Guide*.

2.1.112 `--[no_]strict`

This option enforces or relaxes strict C or strict C++, depending on the choice of source language used.

When `--strict` is selected:

- features that conflict with ISO C or ISO C++ are disabled
- error messages are returned when nonstandard features are used.

Default

The default is `--no_strict`.

Usage

`--strict` enforces compliance with:

- | | |
|----------------|---|
| ISO C90 | <ul style="list-style-type: none"> • ISO/IEC 9899:1990, the 1990 International Standard for C. • ISO/IEC 9899 AM1, the 1995 Normative Addendum 1. |
| ISO C99 | ISO/IEC 9899:1999, the 1999 International Standard for C. |
| ISO C++ | ISO/IEC 14822:2003, the 2003 International Standard for C++. |

Errors

When `--strict` is in force and a violation of the relevant ISO standard occurs, the compiler issues an error message.

The severity of diagnostic messages can be controlled in the usual way.

Example

```
void foo(void)
{
    long long i; /* okay in non-strict C90 */
}
```

Compiling this code with `--strict` generates an error.

See also

- `--c90` on page 2-13
- `--c99` on page 2-13
- `--cpp` on page 2-14
- `--gnu` on page 2-44
- `--strict_warnings`
- *Source language modes* on page 1-3 in the *Compiler User Guide*.

2.1.113 --strict_warnings

This option enforces strict C or strict C++, depending on the choice of source language used.

The `--strict_warnings` option behaves similarly to `--strict`, except that the compiler issues warning messages in place of error messages when nonstandard features are used.

Errors

When `--strict_warnings` is in force and a violation of the relevant ISO standard occurs, the compiler normally issues a warning message.

The severity of diagnostic messages can be controlled in the usual way.

————— Note —————

In some cases, the compiler issues an error message instead of a warning when it detects something that is strictly illegal, and terminates the compilation. For example:

```
#ifdef $Super$
extern void $Super$__aeabi_idiv0(void); /* intercept __aeabi_idiv0 */
#endif
```

Compiling this code with `--strict_warnings` generates an error.

Example

```
void foo(void)
{
    long long i; /* okay in non-strict C90 */
}
```

Compiling this code with `--strict_warnings` generates a warning message.

Compilation continues, even though the expression **long long** is strictly illegal.

See also

- *Source language modes* on page 1-3
- `--c90` on page 2-13
- `--c99` on page 2-13
- `--cpp` on page 2-14
- `--gnu` on page 2-44
- `--[no_]strict` on page 2-84.

2.1.114 --sys_include

This option removes the current place from the include search path.

Quoted include files are treated in a similar way to angle-bracketed include files, except that quoted include files are always searched for first in the directories specified by `-I`, and angle-bracketed include files are searched for first in the `-J` directories.

See also

- `-I dir[,dir,...]` on page 2-46
- `-J dir[,dir,...]` on page 2-51
- `--kandr_include` on page 2-52
- `--preinclude=filename` on page 2-75
- *The current place* on page 2-13 in the *Compiler User Guide*
- *The search path* on page 2-14 in the *Compiler User Guide*.

2.1.115 --thumb

This option configures the compiler to target the Thumb instruction set.

Default

This is the default option for targets that do not support the ARM instruction set.

See also

- `--arm` on page 2-7
- `#pragma arm` on page 4-50
- `#pragma thumb` on page 4-62
- *Specifying the target processor or architecture* on page 2-22 in the *Compiler User Guide*
- *Selecting the target CPU* on page 4-3 in the *Compiler User Guide*.

2.1.116 -Uname

This option removes any initial definition of the macro *name*.

The macro *name* can be either:

- a predefined macro
- a macro specified using the `-D` option.

Note

Not all compiler predefined macros can be undefined.

Syntax

`-Uname`

Where:

name is the name of the macro to be undefined.

Usage

Specifying `-Uname` has the same effect as placing the text `#undef name` at the head of each source file.

Restrictions

The compiler defines and undefines macros in the following order:

1. compiler predefined macros
2. macros defined explicitly, using `-Dname`
3. macros explicitly undefined, using `-Uname`.

See also

- `-C` on page 2-13
- `-Dname[(parm-list)][=def]` on page 2-19
- `-E` on page 2-33
- `-M` on page 2-60
- *Compiler predefines* on page 4-103.

2.1.117 `--[no_]unaligned_access`

This option enables or disables unaligned accesses to data on ARM architecture-based processors.

Default

The default is `--unaligned_access` on ARM-architecture based processors that support unaligned accesses to data. This includes:

- all ARMv6 architecture-based processors
- ARMv7-A and ARMv7-R architecture-based processors.

The default is `--no_unaligned_access` on ARM-architecture based processors that do not support unaligned accesses to data. This includes:

- all pre-ARMv6 architecture-based processors
- ARMv7-M architecture-based processors.

Usage

`--unaligned_access`

Use `--unaligned_access` on processors that support unaligned accesses to data, for example `--cpu=ARM1136J-S`, to speed up accesses to packed structures.

To enable unaligned support, you must:

- Clear the A bit, bit 1, of CP15 register 1 in your initialization code.
- Set the U bit, bit 22, of CP15 register 1 in your initialization code.
The initial value of the U bit is determined by the **UBITINIT** input to the core.

The RVCT libraries include special versions of certain library functions designed to exploit unaligned accesses. When unaligned access support is enabled, the RVCT tools use these library functions to take advantage of unaligned accesses.

--no_unaligned_access

Use `--no_unaligned_access` to disable the generation of unaligned word and halfword accesses on ARMv6 processors.

To enable modulo four-byte alignment checking on an ARMv6 target without unaligned accesses, you must:

- Set the A bit, bit 1, of CP15 register 1 in your initialization code.
- Set the U bit, bit 22, of CP15 register 1 in your initialization code.
The initial value of the U bit is determined by the **UBITINIT** input to the core.

Note

Unaligned doubleword accesses, for example unaligned accesses to **long long** integers, are not supported by ARM processor cores. Doubleword accesses must be either eight-byte or four-byte aligned.

The compiler does not provide support for modulo eight-byte alignment checking. That is, the configuration U = 0, A = 1 in CP15 register 1 is not supported by the compiler, or more generally, by the RVCT toolset.

The RVCT libraries include special versions of certain library functions designed to exploit unaligned accesses. To prevent these enhanced library functions being used when unaligned access support is disabled, you need to specify `--no_unaligned_access` on both the compiler command line and the assembler command line when compiling a mixture of C and C++ source files and assembly language source files.

Restrictions

Code compiled for processors supporting unaligned accesses to data can run correctly only if the choice of alignment support in software matches the choice of alignment support on the processor core.

See also

- `--cpu=name` on page 2-15
- *Command syntax* on page 3-2 in the *Assembler Guide*
- *Alignment support* on page 4-43 in the *Compiler User Guide*.

2.1.118 --use_pch=filename

This option instructs the compiler to use a PCH file with the specified filename as part of the current compilation.

This option takes precedence if you include `--pch` on the same command line.

Syntax

`--use_pch=filename`

Where:

filename is the PCH file to be used as part of the current compilation.

Restrictions

The effect of this option is negated if you include `--create_pch=filename` on the same command line.

Errors

If the specified file does not exist, or is not a valid PCH file, then the compiler generates an error.

See also

- `--create_pch=filename` on page 2-18
- `--pch` on page 2-70
- `--pch_dir=dir` on page 2-71
- `--[no_]pch_messages` on page 2-72
- `--[no_]pch_verbose` on page 2-72
- *Precompiled header files* on page 2-16 in the *Compiler User Guide*.

2.1.119 `--[no_]using_std`

This option enables or disables implicit use of the `std` namespace when standard header files are included in C++.

———— Note ————

This option is provided only as a migration aid for legacy source code that does not conform to the C++ standard. Its use is not recommended.

Mode

This option is effective only if the source language is C++.

Default

The default is `--no_using_std`.

See also

- *Namespaces* on page 5-16.

2.1.120 --[no_]vectorize

This option enables or disables the generation of NEON vector instructions directly from C or C++ code.

Default

The default is `--no_vectorize`.

Restrictions

The following options must be specified for loops to vectorize:

- `--cpu=name` Target processor must have NEON capability.
- `-Otime` Type of optimization to reduce execution time.
- `-Onum` Level of optimization. One of the following must be used:
 - `-O2` High optimization. This is the default.
 - `-O3` Maximum optimization.

Note

NEON is an implementation of the ARM Advanced *Single Instruction, Multiple Data* (SIMD) extension.

A separate *FLEXnet* license is needed to enable the use of vectorization.

Example

```
armcc --vectorize --cpu=Cortex-A8 -O3 -Otime -c file.c
```

See also

- `--cpu=name` on page 2-15
- `-Onum` on page 2-66
- `-Otime` on page 2-69

- The *NEON Vectorizing Compiler Guide*.

2.1.121 --[no_]vfe

This option enables or disables *Virtual Function Elimination* (VFE) in C++.

VFE enables unused virtual functions to be removed from code. When VFE is enabled, the compiler places the information in special sections with the prefix `.arm_vfe_`. These sections are ignored by linkers that are not VFE-aware, because they are not referenced by the rest of the code. Therefore, they do not increase the size of the executable. However, they increase the size of the object files.

Mode

This option is effective only if the source language is C++.

Default

The default is `--vfe`, except for the case where legacy object files compiled with a pre-RVCT v2.1 compiler do not contain VFE information.

See also

- *Calling a pure virtual function* on page C-3
- *Unused function elimination* on page 3-12 in the *Linker Guide*.

2.1.122 --via=filename

This option instructs the compiler to read additional command-line options from a specified file. The options read from the file are added to the current command line.

Via commands can be nested within via files.

Syntax

`--via=filename`

Where:

filename is the name of a via file containing options to be included on the command line.

Example

Given a source file `main.c`, a via file `apcs.txt` containing the line:

```
--apcs=/rwpi --no_lower_rwpi --via=L_apcs.txt
```

and a second via file `L_apcs.txt` containing the line:

```
-L--rwpi -L--callgraph
```

compiling `main.c` with the command line:

```
armcc main.c -L-o"main.axf" --via=apcs.txt
```

compiles `main.c` using the command line:

```
armcc --no_lower_rwpi --apcs=/rwpi -L--rwpi -L--callgraph -L-o"main.axf" main.c
```

See also

- [Appendix A Via File Syntax](#)
- [Reading compiler options from a file](#) on page 2-10 in the *Compiler User Guide*.

2.1.123 --[no_]vla

This option enables or disables support for variable length arrays.

Default

C90 and Standard C++ do not support variable length arrays by default. Select the option `--vla` to enable support for variable length arrays in C90 or Standard C++.

Variable length arrays are supported both in Standard C and the GNU compiler extensions. The option `--vla` is implicitly selected either when the source language is C99 or the option `--gnu` is specified.

Example

```
size_t arr_size(int n)
{
    char array[n];           // variable length array, dynamically allocated

    return sizeof array;    // evaluated at runtime
}
```

See also

- `--c90` on page 2-13
- `--c99` on page 2-13
- `--cpp` on page 2-14
- `--gnu` on page 2-44.

2.1.124 --vsn

This option displays the version information and the license details.

See also

- `--help` on page 2-45.

2.1.125 -W

This option instructs the compiler to suppress all warning messages.

See also

- `--[no_]brief_diagnostics` on page 2-10
- `--diag_error=tag[,tag,...]` on page 2-26
- `--diag_remark=tag[,tag,...]` on page 2-27
- `--diag_style={arm|ide|gnu}` on page 2-27
- `--diag_suppress=tag[,tag,...]` on page 2-28
- `--diag_warning=tag[,tag,...]` on page 2-30
- `--errors=filename` on page 2-34
- `--remarks` on page 2-78
- `--[no_]wrap_diagnostics` on page 2-96

2.1.126 --wchar16

This option changes the type of `wchar_t` to **unsigned short**.

Selecting this option modifies both the type of the defined type `wchar_t` in C and the type of the native type **`wchar_t`** in C++. It also affects the values of `WCHAR_MIN` and `WCHAR_MAX`.

Default

The compiler assumes `--wchar16` unless `--wchar32` is explicitly specified.

See also

- `--wchar32`
- *Predefined macros* on page 4-103.

2.1.127 --wchar32

This option changes the type of `wchar_t` to **unsigned int**.

Selecting this option modifies both the type of the defined type `wchar_t` in C and the type of the native type `wchar_t` in C++. It also affects the values of `WCHAR_MIN` and `WCHAR_MAX`.

Default

The compiler assumes `--wchar16` unless `--wchar32` is explicitly specified.

See also

- `--wchar16` on page 2-94
- *Predefined macros* on page 4-103.

2.1.128 `--workdir=directory`

This option enables you to provide a working directory for a project template.

————— Note —————

Project templates only require working directories if they include files, for example, RVD configuration files.

Syntax

`--workdir=directory`

Where:

directory is the name of the project directory.

Restrictions

If you specify a project working directory using `--workdir`, then you must specify a project file using `--project`.

Errors

An error message is produced if you try to use `--project` without `--workdir` and `--workdir` is required.

See also

- `--[no_]project=filename` on page 2-74
- `--reinitialize_workdir` on page 2-77.

2.1.129 --[no_]wrap_diagnostics

This option enables or disables the wrapping of error message text when it is too long to fit on a single line.

Default

The default is `--no_wrap_diagnostics`.

See also

- `--[no_]brief_diagnostics` on page 2-10
- `--diag_error=tag[,tag,...]` on page 2-26
- `--diag_remark=tag[,tag,...]` on page 2-27
- `--diag_style={arm|ide|gnu}` on page 2-27
- `--diag_suppress=tag[,tag,...]` on page 2-28
- `--diag_warning=tag[,tag,...]` on page 2-30
- `--errors=filename` on page 2-34
- `--remarks` on page 2-78
- `-W` on page 2-94
- Chapter 5 *Diagnostic Messages* in the *Compiler User Guide*.

Chapter 3

Language Extensions

This chapter describes the language extensions supported by the ARM compiler, and includes:

- *Preprocessor extensions* on page 3-2
- *C99 language features available in C90* on page 3-5
- *C99 language features available in C++ and C90* on page 3-7
- *Standard C language extensions* on page 3-10
- *Standard C++ language extensions* on page 3-15
- *Standard C and standard C++ language extensions* on page 3-19
- *GNU language extensions* on page 3-25.

For additional reference material on the ARM compiler see also:

- Appendix B *Standard C Implementation Definition*
- Appendix C *Standard C++ Implementation Definition*
- Appendix D *C and C++ Compiler Implementation Limits*.

3.1 Preprocessor extensions

The compiler supports several extensions to the preprocessor, including the `#assert` preprocessing extensions of System V release 4.

3.1.1 `#assert`

The `#assert` preprocessing extensions of System V release 4 are permitted. These enable definition and testing of predicate names.

Such names are in a namespace distinct from all other names, including macro names.

Syntax

`#assert name`

`#assert name[(token-sequence)]`

Where:

name is a predicate name

token-sequence is an optional sequence of tokens.

If the token sequence is omitted, *name* is not given a value.

If the token sequence is included, *name* is given the value *token-sequence*.

Example

A predicate name defined using `#assert` can be tested in a `#if` expression, for example:

```
#if #name(token-sequence)
```

This has the value 1 if a `#assert` of the name *name* with the token-sequence *token-sequence* has appeared, and 0 otherwise. A given predicate can be given more than one value at a given time.

See also

- `#unassert` on page 3-3.

3.1.2 `#include_next`

This preprocessor directive is a variant of the `#include` directive. It searches for the named file only in the directories on the search path that follow the directory where the current source file is found, that is, the one containing the `#include_next` directive.

Note

This preprocessor directive is a GNU compiler extension that is supported by the ARM compiler.

3.1.3 `#unassert`

A predicate name can be deleted using a `#unassert` preprocessing directive.

Syntax

```
#unassert name
```

```
#unassert name[(token-sequence)]
```

Where:

name is a predicate name

token-sequence is an optional sequence of tokens.

If the token sequence is omitted, all definitions of *name* are removed.

If the token sequence is included, only the indicated definition is removed. All other definitions are left intact.

See also

- `#assert` on page 3-2.

3.1.4 `#warning`

The preprocessing directive `#warning` is supported. Like the `#error` directive, this produces a user-defined warning at compilation time. However, it does not halt compilation.

Restrictions

The `#warning` directive is not available if the `--strict` option is specified. If used, it produces an error.

See also

- `--[no_]strict` on page 2-84.

3.2 C99 language features available in C90

The compiler supports numerous extensions to the ISO C90 standard, for example, C99-style `//` comments.

These extensions are available if the source language is C90 and you are compiling in non strict mode.

These extensions are not available if the source language is C90 and the compiler is restricted to compiling strict C90 using the `--strict` compiler option.

Note

Language features of Standard C and Standard C++, for example C++-style `//` comments, might be similar to the C90 language extensions described in this section. Such features continue to remain available if you are compiling strict Standard C or strict Standard C++ using the `--strict` compiler option.

3.2.1 `//` comments

The character sequence `//` starts a one line comment, just as in C99 or C++.

`//` comments in C90 have the same semantics as `//` comments in C99.

Example

```
// this is a comment
```

See also

- *New features of C99* on page 1-5 in the *Compiler User Guide*.

3.2.2 Subscripting struct

In C90, arrays that are not lvalues still decay to pointers, and can be subscripted. However, you must not modify or use them after the next sequence point, and you must not apply the unary `&` operator to them. Arrays of this kind can be subscripted in C90, but they do not decay to pointers outside C99 mode.

Example

```
struct Subscripting_Struct
{
    int a[4];
};
```

```
extern struct Subscripting_Struct Subscripting_0(void);

int Subscripting_1 (int index)
{
    return Subscripting_0().a[index];
}
```

3.2.3 Flexible array members

The last member of a **struct** can have an incomplete array type. The last member must not be the only member of the **struct**, otherwise the **struct** is zero in size.

Example

```
typedef struct
{
    int len;
    char p[]; // incomplete array type, for use in a malloced data structure
} str;
```

See also

- *New features of C99* on page 1-5 in the *Compiler User Guide*.

3.3 C99 language features available in C++ and C90

The compiler supports numerous extensions to the ISO C++ standard and to the C90 language, for example, function prototypes that override old-style non prototype definitions.

These extensions are available if:

- the source language is C++ and you are compiling in non strict mode
- the source language is C90 and you are compiling in non strict mode.

These extensions are not available if:

- the source language is C++ and the compiler is restricted to compiling strict C90 using the `--strict` compiler option.
- the source language is C90 and the compiler is restricted to compiling strict Standard C using the `--strict` compiler option.

Note

Language features of Standard C, for example **long long** integers, might be similar to the language extensions described in this section. Such features continue to remain available if you are compiling strict standard C++ or strict C90 using the `--strict` compiler option.

3.3.1 Variadic macros

In C90 and C++ you can declare a macro to accept a variable number of arguments.

The syntax for declaring a variadic macro in C90 and C++ follows the C99 syntax for declaring a variadic macro, unless the option `--gnu` is selected. If the option `--gnu` is specified, the syntax follows GNU syntax for variadic macros.

Example

```
#define debug(format, ...) fprintf (stderr, format, __VA_ARGS__)

void variadic_macros(void)
{
    debug ("a test string is printed out along with %x %x %x\n", 12, 14, 20);
}
```

See also

- `--gnu` on page 2-44

- *New features of C99* on page 1-5 in the *Compiler User Guide*.

3.3.2 long long

The ARM compiler supports 64-bit integer types through the type specifiers **long long** and **unsigned long long**. They behave analogously to **long** and **unsigned long** with respect to the usual arithmetic conversions. `__int64` is a synonym for **long long**.

Integer constants can have:

- an `ll` suffix to force the type of the constant to **long long**, if it fits, or to **unsigned long long** if it does not fit
- a `ull` or `llu` suffix to force the type of the constant to **unsigned long long**.

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

Also, a plain integer constant is of type **long long** or **unsigned long long** if its value is large enough. There is a warning message from the compiler indicating the change. For example, in strict 1990 ISO Standard C 2147483648 has type **unsigned long**. In ARM C and C++ it has the type **long long**. One consequence of this is the value of an expression such as:

```
2147483648 > -1
```

This expression evaluates to 0 in strict C and C++, and to 1 in ARM C and C++.

The **long long** types are accommodated in the usual arithmetic conversions.

See also

- `__int64` on page 4-9.

3.3.3 restrict

The **restrict** keyword is a C99 feature that enables you to ensure that different object pointer types and function parameter arrays do not point to overlapping regions of memory. Therefore, the compiler performs optimizations that can be otherwise prevented because of possible aliasing.

Restrictions

To enable the **restrict** keyword in C90 or C++, you must specify the `--restrict` option.

The keywords `__restrict` and `__restrict__` are supported as synonyms for **restrict** and are always available, regardless of whether `--restrict` is specified.

Example

```
void copy_array(int n, int *restrict a, int *restrict b)
{
    while (n-- > 0)
        *a++ = *b++;
}
```

See also

- `--[no_]restrict` on page 2-79
- *New features of C99* on page 1-5 in the *Compiler User Guide*.

3.3.4 Hex floats

C90 and C++ support floating-point numbers that can be written in hexadecimal format.

Example

```
float hex_floats(void)
{
    return 0x1.fp3;    // 1.55e1
}
```

See also

- *New features of C99* on page 1-5 in the *Compiler User Guide*.

3.4 Standard C language extensions

The compiler supports numerous extensions to the ISO C99 standard, for example, function prototypes that override old-style non prototype definitions.

These extensions are available if:

- the source language is C99 and you are compiling in non strict mode
- the source language is C90 and you are compiling in non strict mode.

None of these extensions is available if:

- the source language is C90 and the compiler is restricted to compiling strict C90 using the `--strict` compiler option.
- the source language is C99 and the compiler is restricted to compiling strict Standard C using the `--strict` compiler option.
- the source language is C++.

3.4.1 Constant expressions

Extended constant expressions are supported in initializers. The following examples show the compiler behavior for the default, `--strict_warnings`, and `--strict` compiler modes.

Example 1, assigning the address of variable

Your code might contain constant expressions that assign the address of a variable at file scope, for example:

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

When compiling for C, this produces the following behavior:

- In default mode a warning is produced.
- In `--strict_warnings` mode a warning is produced.
- In `--strict` mode, an error is produced.

Example 2, constant value initializers

The compiler behavior when you have expressions that include constant values in C code is summarized in the following example:

```
extern int  const c = 10;          /* Std   RVCT v3.1 */
extern int  const x = c + 10;      /* ok    ok    */
/* error  ext    */
```

```

static int      y = c + 10;          /* error    ext    */
static int const z = c + 10;          /* error    ext    */
extern int *const cp = (int*)0x100;   /* ok       ok     */
extern int *const xp = cp + 0x100;    /* error    ext    */
static int *      yp = cp + 0x100;    /* error    ext    */
static int *const zp = cp + 0x100;    /* error    ext    */

```

This indicates the behavior defined by the ISO C Standard, Std, and the behavior in RVCT:

- ok indicates that the statement is accepted in all C modes.
- ext is an extension to the ISO C Standard. The behavior depends on the strict mode used when compiling C:

Non strict

Accepted, without a warning.

--strict_warnings

Accepted, but gives a warning.

--strict

Conforms to the ISO C Standard, but gives an error.

See also

- `--[no_]extended_initializers` on page 2-36
- `--[no_]strict` on page 2-84
- `--strict_warnings` on page 2-85.

3.4.2 Array and pointer extensions

The following array and pointer extensions are supported:

- Assignment and pointer differences are permitted between pointers to types that are interchangeable but not identical, for example, **unsigned char *** and **char ***. This includes pointers to same-sized integral types, typically, **int *** and **long ***. A warning is issued, for example:
#513-D: a value of type "char *" cannot be assigned to an entity of type "unsigned char *"
Assignment of a string constant to a pointer to any kind of character is permitted without a warning.
- Assignment of pointer types is permitted in cases where the destination type has added type qualifiers that are not at the top level, for example, assigning **int **** to **const int ****. Comparisons and pointer difference of such pairs of pointer types are also permitted. A warning is issued, for example:

#42-D: operand types are incompatible ("const int **" and "int **")

- In operations on pointers, a pointer to void is always implicitly converted to another type if necessary. Also, a null pointer constant is always implicitly converted to a null pointer of the right type if necessary. In ISO C, some operators permit these, and others do not.
- Pointers to different function types can be assigned or compared for equality (==) or inequality (!=) without an explicit type cast. A warning or error is issued, for example:
#42-D: operand types are incompatible ("int (*)(char *)" and "unsigned int (*)(char *)")
This extension is prohibited in C++ mode.
- A pointer to **void** can be implicitly converted to, or from, a pointer to a function type.
- In an initializer, a pointer constant value can be cast to an integral type if the integral type is big enough to contain it.
- A non lvalue array expression is converted to a pointer to the first element of the array when it is subscripted or similarly used.

3.4.3 Block scope function declarations

Two extensions to block scope function declarations are supported:

- a block-scope function declaration also declares the function name at file scope
- a block-scope function declaration can have static storage class, thereby causing the resulting declaration to have internal linkage by default.

Example

```
void f1(void)
{
    static void g(void); /* static function declared in local scope */
                          /* use of static keyword is illegal in strict ISO C */
}

void f2(void)
{
    g();                  /* uses previous local declaration */
}

static void g(int i)
{ } /* error - conflicts with previous declaration of g */
```

3.4.4 Dollar signs in identifiers

Dollar (\$) signs are permitted in identifiers.

Note

When compiling with the `--strict` option, you can use the `--dollar` command-line option to permit dollar signs in identifiers.

Example

```
#define DOLLAR$
```

See also

- `--[no_]dollar` on page 2-32
- `--[no_]strict` on page 2-84.

3.4.5 Top-level declarations

A C input file can contain no top-level declarations.

Errors

A remark is issued if a C input file contains no top-level declarations.

Note

Remarks are not displayed by default. To see remark messages, use the compiler option `--remarks`.

See also

- `--remarks` on page 2-78.

3.4.6 Benign redeclarations

Benign redeclarations of **typedef** names are permitted. That is, a **typedef** name can be redeclared in the same scope as the same type.

Example

```
typedef int INT;
typedef int INT; /* redeclaration */
```

3.4.7 External entities

External entities declared in other scopes are visible.

Errors

The compiler generates a warning if an external entity declared in another scope is visible.

Example

```
void f1(void)
{
    extern void f();
}

void f2(void)
{
    f(); /* Out of scope declaration */
}
```

3.4.8 Function prototypes

The compiler recognizes function prototypes that override old-style non prototype definitions that appear at a later position in your code, for example:

Errors

The compiler generates a warning message if you use old-style function prototypes.

Example

```
int function_prototypes(char);

// Old-style function definition.
int function_prototypes(x)
    char x;
{
    return x == 0;
}
```

3.5 Standard C++ language extensions

The compiler supports numerous extensions to the ISO C++ standard, for example, qualified names in the declaration of class members.

These extensions are available if the source language is C++ and you are compiling in non strict mode.

These extensions are not available if the source language is C++ and the compiler is restricted to compiling strict Standard C++ using the `--strict` compiler option.

3.5.1 ? operator

A `?` operator whose second and third operands are string literals or wide string literals can be implicitly converted to `char *` or `wchar_t *`. In C++ string literals are `const`. There is an implicit conversion that enables conversion of a string literal to `char *` or `wchar_t *`, dropping the `const`. That conversion, however, applies only to simple string literals. Permitting it for the result of a `?` operation is an extension.

Example

```
char *p = x ? "abc" : "def";
```

3.5.2 Declaration of a class member

A qualified name can be used in the declaration of a class member.

Errors

A warning is issued if a qualified name is used in the declaration of a class member.

Example

```
struct A
{
    int A::f(); // is the same as int f();
};
```

3.5.3 friend

A **friend** declaration for a **class** can omit the **class** keyword.

Access checks are not carried out on **friend** declarations by default. Use the `--strict` command-line option to force access checking.

Example

```
class B;
class A
{
    friend B; // is the same as "friend class B"
};
```

See also

- `--[no_]strict` on page 2-84.

3.5.4 Read/write constants

A 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 with `--apcs /rwp`. If you recompile existing code with this option, you must change the linkage specification for external constants that are dynamically initialized or have mutable members.

Compiling C++ with the `--apcs /rwp` option deviates from the ISO C++ Standard. The declarations in Example 3-1 assume that `x` is in a read-only segment.

Example 3-1 External access

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

Dynamic initialization of `x` including user-defined constructors is not possible for constants and `T` cannot contain mutable members. The new linkage specification in Example 3-2 on page 3-17 declares that `x` is in a read/write segment even if it is initialized with a constant. Dynamic initialization of `x` is permitted and `T` can contain mutable members. The definitions of `x`, `y`, and `z` in another file must have the same linkage specifications.

Example 3-2 Linkage specification

```

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 3-3 produces a compile error.

Example 3-3 Compiler error

```

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

```

Note

Because C does not have the linkage specifications, you cannot use a **const** object declared in C++ as `extern "C++:read/write"` from C.

See also

- `--apcs=qualifer...qualifier` on page 2-4.

3.5.5 Scalar type constants

Constants of scalar type can be defined within classes. This is an old form. The modern form uses an initialized static data member.

Errors

A warning is issued if you define a member of constant integral type within a class.

Example

```
class A
{
    const int size = 10; // must be static
    int a[size];
};
```

3.5.6 Type conversions

Type conversion between a pointer to an extern "C" function and a pointer to an extern "C++" function is permitted.

Example

```
extern "C" void f();    // f's type has extern "C" linkage
void (*pf)() = &f;     // pf points to an extern "C++" function
                      // error unless implicit conversion is allowed
```

3.6 Standard C and standard C++ language extensions

The compiler supports numerous extensions to both the ISO C99 and the ISO C++ Standards, such as various integral type extensions, various floating-point extensions, hexadecimal floating-point constants, and anonymous classes, structures, and unions.

These extensions are available if:

- the source language is C++ and you are compiling in non strict mode
- the source language is C99 and you are compiling in non strict mode
- the source language is C90 and you are compiling in non strict mode.

These extensions are not available if:

- the source language is C++ and the compiler is restricted to compiling strict C++ using the `--strict` compiler option.
- the source language is C99 and the compiler is restricted to compiling strict Standard C using the `--strict` compiler option.
- the source language is C90 and the compiler is restricted to compiling strict C90 using the `--strict` compiler option.

3.6.1 Address of a register variable

The address of a variable with **register** storage class can be taken.

Errors

The compiler generates a warning if you take the address of a variable with **register** storage class.

Example

```
void foo(void)
{
    register int i;
    int *j = &i;
}
```

3.6.2 Arguments to functions

Default arguments can be specified for function parameters other than those of a top-level function declaration. For example, they are accepted on typedef declarations and on pointer-to-function and pointer-to-member-function declarations.

3.6.3 Anonymous classes, structures and unions

Anonymous classes, structures, and unions are supported as an extension. Anonymous structures and unions are supported in C and C++.

Anonymous unions are available by default in C++. However, you must specify the `anon_unions` pragma if you want to use:

- anonymous unions and structures in C
- anonymous classes and structures in C++.

An anonymous union can be introduced into a containing class by a **typedef** name. Unlike a true anonymous union, it does not have to be declared directly. For example:

```
typedef union
{
    int i, j;
} U;           // U identifies a reusable anonymous union.

#pragma anon_unions

class A
{
    U;           // Okay -- references to A::i and A::j are allowed.
};
```

The extension also enables anonymous classes and anonymous structures, as long as they have no C++ features. For example, no static data members or member functions, no non public members, and no nested types (except anonymous classes, structures, or unions) are allowed in anonymous classes and anonymous structures. For example:

```
#pragma anon_unions

struct A
{
    struct
    {
        int i, j;
    };           // Okay -- references to A::i and A::j are allowed.
};
```

See also

- *Unnamed fields* on page 3-31
- *#pragma [no_]anon_unions* on page 4-49.

3.6.4 Assembler labels

Assembler labels specify the assembler name to use for a C symbol. For example, you might have assembler code and C code that uses the same symbol name, such as `counter`. Therefore, you can export a different name to be used by the assembler:

```
int counter __asm__("counter_v1") = 0;
```

This exports the symbol `counter_v1` and not the symbol `counter`.

See also

- `__asm` on page 4-5.

3.6.5 Empty declaration

An empty declaration, that is a semicolon with nothing before it, is permitted.

Example

```
; // do nothing
```

3.6.6 Hexadecimal floating-point constants

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

Syntax

The syntax for specifying floating-point constants as IEEE bit patterns is:

<code>0f_n</code>	Interpret an 8-digit hex number <i>n</i> as a float constant. There must be exactly eight digits.
<code>0d_nn</code>	Interpret a 16-digit hex number <i>nn</i> as a double constant. There must be exactly 16 digits.

3.6.7 Incomplete enums

Forward declarations of enums are supported.

Example

```
enum Incomplete_Enums_0;

int Incomplete_Enums_2 (enum Incomplete_Enums_0 * passon)
```

```

{
    return 0;
}

int Incomplete_Enums_1 (enum Incomplete_Enums_0 * passon)
{
    return Incomplete_Enums_2(passon);
}

enum Incomplete_Enums_0 { ALPHA, BETA, GAMMA };

```

3.6.8 Integral type extensions

In an integral constant expression, an integral constant can be cast to a pointer type and then back to an integral type.

3.6.9 Label definitions

In Standard C and Standard C++, a statement must follow a label definition. In C and C++, a label definition can be followed immediately by a right brace.

Errors

The compiler generates a warning if a label definition is followed immediately by a right brace.

Example

```

void foo(char *p)
{
    if (p)
    {
        /* ... */
label:
    }
}

```

3.6.10 Long float

long float is accepted as a synonym for **double**.

3.6.11 Non static local variables

Non static local variables of an enclosing function can be referenced in a non evaluated expression, for example, a sizeof expression inside a local class. A warning is issued.

3.6.12 Structure, union, enum, and bitfield extensions

The following structure, union, enum, and bitfield extensions are supported:

- In C, the element type of a file-scope array can be an incomplete **struct** or **union** type. The element type must be completed before its size is needed, for example, if the array is subscripted. If the array is not **extern**, the element type must be completed by the end of the compilation.
- The final semicolon preceding the closing brace **}** of a **struct** or **union** specifier can be omitted. A warning is issued.
- An initializer expression that is a single value and is used to initialize an entire static array, **struct**, or **union**, does not have to be enclosed in braces. ISO C requires the braces.
- An extension is supported to enable constructs similar to C++ anonymous unions, including the following:
 - not only anonymous unions but also anonymous structs are permitted. The members of anonymous structs are promoted to the scope of the containing **struct** and looked up like ordinary members.
 - they can be introduced into the containing **struct** by a **typedef** name. That is, they do not have to be declared directly, as is the case with true anonymous unions.
 - a tag can be declared but only in C mode.

To enable support for anonymous structures and unions, you must use the `anon_unions` pragma.

- An extra comma is permitted at the end of an **enum** list but a remark is issued.
- **enum** tags can be incomplete. You can define the tag name and resolve it later, by specifying the brace-enclosed list.
- The values of enumeration constants can be given by expressions that evaluate to unsigned quantities that fit in the **unsigned int** range but not in the **int** range. For example:


```
/* When ints are 32 bits: */
enum a { w = -2147483648 }; /* No error */
enum b { x = 0x80000000 }; /* No error */
enum c { y = 0x80000001 }; /* No error */
enum d { z = 2147483649 }; /* Error */
```
- Bit fields can have base types that are **enum** types or integral types besides **int** and **unsigned int**.

See also

- *Pragmas* on page 4-49
- *Structure, union, enum, and bitfield extensions* on page 3-23
- *New features of C99* on page 1-5 in the *Compiler User Guide*.

3.7 GNU language extensions

This section describes GNU compiler extensions that are supported by the ARM compiler. These extensions are supported only in GNU mode, that is, when you compile your source code with the `--gnu` option. See *Language compliance* on page 1-6 and `--gnu` on page 2-44 for more information.

Note

Not all GNU compiler extensions are supported for all languages. For example, extended pointer arithmetic is not supported for C++.

For more information on the use of the GNU extensions, see the GNU compiler documentation online at <http://gcc.gnu.org>.

For additional reference material on the ARM compiler see also:

- Appendix B *Standard C Implementation Definition*
- Appendix C *Standard C++ Implementation Definition*
- Appendix D *C and C++ Compiler Implementation Limits*.

3.7.1 Alternate keywords

The compiler recognizes alternate keywords of the form `__keyword__`. These alternate keywords have the same behavior as the original keywords.

Example

```
__const__ int pi = 3.14; // same as const int pi = 3.14
```

3.7.2 asm keyword

This keyword is a synonym for the `__asm` keyword.

Mode

Supported in GNU mode for C90 and C99 only.

See also

- `__asm` on page 4-5.

3.7.3 Case ranges

You can specify ranges of values in **switch** statements.

Example

```

int Case_Ranges_0(int arg)
{
    int aLocal;
    int bLocal =arg;
    switch (bLocal)
    {
        case 0 ... 10:
            aLocal= 1;
            break;
        case 11 ... 100:
            aLocal =2;
            break;
        default:
            aLocal=-1;
    }
    return aLocal;
}

```

3.7.4 Cast of a union

A cast to a union type is similar to other casts, except that the type specified is a union type. You can specify the type either with a union tag or with a typedef name.

Mode

Supported in GNU mode for C90 and C99 only.

Example

```

typedef union
{
    double d;
    int i;
} foo_t;

int Cast_to_Union_0(int a, double b)
{
    foo_t u;
    if (a>100)
        u = (foo_t) a ; // automatically equivalent to u.i=a;
    else
        u = (foo_t) b ; // automatically equivalent to u.d=b;
    return u.i;
}

```

3.7.5 Character escape sequences

In strings, the escape sequence ‘\e’ is accepted for the escape character <ESC> (ASCII 27).

Example

```
void foo(void)
{
    printf("Escape sequence is: \e\n");
}
```

3.7.6 Compound literals

As in C99, compound literals are supported. All compound literals are lvalues.

Example

```
int y[] = (int []) {1, 2, 3}; // error in strict C99, okay in C99 --gnu
int z[] = (int [3]) {1};
```

Mode

Supported in GNU mode for C90 and C99 only.

———— Note —————

Compound literals can also be used as initializers in C99. However, the compiler is more relaxed about which compound literals it accepts as initializers in GNU mode than it is when compiling C99 source code.

3.7.7 Designated inits

As in C99, designated initializers are supported.

Example

```
int a[6] = { [4] = 29, [2] = 15 };
int b[6] = { 0,0,15,0,29,0 }; // a[] is equivalent to b[]
```

Mode

Supported in GNU mode for C90 and C++ only.

See also

- *New features of C99* on page 1-5 in the *Compiler User Guide*.

3.7.8 Conditionals

The middle operand in a conditional statement can be omitted, if the result is to be the same as the test, for example:

```
i ? i : j;
```

This is most useful if the test modifies the value in some way, for example:

```
i++ ? : j;
```

where `i++` comes from a macro. If you write code in this way, then `i++` is evaluated only once.

If the original value of `i` is nonzero, the result is the original value of `i`. Regardless of this, `i` is incremented once.

3.7.9 Initializers

As in standard C++ and ISO C99, the elements of an aggregate initializer for an automatic variable are not required to be constant expressions.

Mode

Supported in GNU mode only for C90.

Example

```
float Initializers_0 (float f, float g)
{
    float beat_freqs[2] = { f-g, f+g };
    float aLocal;
    int i=0;

    for (; i<2; i++)
        aLocal += beat_freqs[i];
    return aLocal;
}
```

3.7.10 Inline

The `inline` function qualifier specifies that the function is to be inlined.

```
static inline foo () {...}
```

foo is used internally to the file, and the symbol is not exported.

```
inline foo() {...}
```

foo is used internally to the file and an out of line version is made available and the name foo exported.

```
extern inline foo () {...}
```

In GNU mode, foo is used internally if it is inlined. If it is not inlined then an external version is referenced rather than using a call to the internal version. Also, the foo symbol is not emitted.

In non-GNU mode, **extern** is ignored and the functionality is the same as **inline** foo() for C++. In C, you must use `__inline`. See *Extern inline functions* on page 5-19 for more information.

Mode

Supported in GNU mode only for C90.

3.7.11 Extended lvalues

The definition of what constitutes an lvalue when looking at comma expressions and `?:` constructs is relaxed in GNU mode. You can use compound expressions, conditional expressions, and casts as follows:

- You can assign a compound expression:

```
(a++, b) += x;
```

This is equivalent to:

```
temp = (a++,b);
b = temp + x
```

- You can get the address of a compound expression `&(a, b)`. This is equivalent to `(a, &b)`.
- You can use conditional expressions, for example:

```
(a ? b : c) = a;
```

This picks b or c as the destination dependent on a.

Mode

Supported in GNU mode only for C90 and C99 only.

3.7.12 Pointer arithmetic

You can perform arithmetic on void pointers and function pointers.

The size of a void type or a function type is defined to be 1.

Mode

Supported in GNU mode for C90 and C99 only.

Errors

The compiler generates a warning if it detects arithmetic on **void** pointers or function pointers.

Example

```
int ptr_arith_0(void)
{
    void * pointer;
    return sizeof *pointer;
}

int ptr_arith_1(void)
{
    static int diff;

    diff = ptr_arith_0 - ptr_arith_1;

    return sizeof ptr_arith_0;
}
```

3.7.13 Statement expressions

Statement expressions enable you to place whole sections of code, including declarations, within braces ({ }) .

The result of a statement expression is the final item in the statement list.

Example

```

int bar(int b, int foo)
{
    if ({
        int y = foo;
        int z;

        if (y > 0) z = y;
        else z = -y;
        z>b;
    })
    b++;
    return b;
}

```

3.7.14 Unnamed fields

When embedding a structure or union within another structure or union, you do not have to name the internal structure. You can access the contents of the unnamed structure without using `.name` to reference it.

Unnamed fields are the same as anonymous unions and structures.

Mode

Supported in GNU mode for C90 and C99 only.

Example

```

struct
{
    int a;
    union
    {
        int b;
        float c;
    };
    int d;
} Unnamed_Fields_0;

int Unnamed_Fields_1()
{
    return Unnamed_Fields_0.b;
}

```

See also

- *Anonymous classes, structures and unions* on page 3-20.

Chapter 4

Compiler-specific Features

This chapter describes the ARM compiler-specific features, and includes:

- *Keywords and operators* on page 4-2
- *__declspec attributes* on page 4-24
- *Function attributes* on page 4-29
- *Type attributes* on page 4-37
- *Variable attributes* on page 4-40
- *Pragmas* on page 4-49
- *Instruction intrinsics* on page 4-63
- *Compiler predefines* on page 4-103.

4.1 Keywords and operators

This section describes the function keywords and operators supported by the ARM compiler armcc.

Table 4-1 lists keywords that are ARM extensions to the C and C++ Standards. Standard C and Standard C++ keywords that do not have behavior or restrictions specific to the ARM compiler are not documented in the table.

Table 4-1 Keyword extensions supported by the ARM compiler

Keywords		
<code>__align</code>	<code>__int64</code>	<code>__svc</code>
<code>__ALIGNOF__</code>	<code>__INTADDR__</code>	<code>__svc_indirect</code>
<code>__asm</code>	<code>__irq</code>	<code>__svc_indirect_r7</code>
<code>__declspec</code>	<code>__packed</code>	<code>__value_in_regs</code>
<code>__forceinline</code>	<code>__pure</code>	<code>__weak</code>
<code>__global_reg</code>	<code>__softfp</code>	<code>__writeonly</code>
<code>__inline</code>	<code>__smc</code>	

4.1.1 __align

The `__align` keyword instructs the compiler to align a variable on an n -byte boundary. `__align` is a storage class modifier. It does not affect the type of the function.

Syntax

`__align( $n$ )`

Where:

- n is the alignment boundary.
 n can take the values 1, 2, 4, or 8.
- The keyword `__align` comes immediately before the variable name.

Usage

`__align(n)` is useful when the normal alignment of the variable being declared is less than *n*. Eight-byte alignment can give a significant performance advantage with VFP instructions.

`__align` can be used in conjunction with **extern** and **static**.

Restrictions

Because `__align` is a storage class modifier, it cannot be used on:

- types, including **typedefs** and structure definitions
- function parameters.

You can only overalign. That is, you can make a two-byte object four-byte aligned but you cannot align a four-byte object at 2 bytes.

Example

```
__align(8) char buffer[128]; /* buffer starts on eight-byte boundary */
```

See also

- `--min_array_alignment=opt` on page 2-62 in the *Compiler Reference Guide*.

4.1.2 `__alignof__`

The `__alignof__` keyword enables you to enquire about the alignment of a type or variable.

————— **Note** —————

This keyword is a GNU compiler extension that is supported by the ARM compiler.

Syntax

```
__alignof__(type)
```

```
__alignof__(expr)
```

Where:

<i>type</i>	is a type
<i>expr</i>	is an lvalue.

Return value

`__alignof__(type)` returns the alignment requirement for the type *type*, or 1 if there is no alignment requirement.

`__alignof__(expr)` returns the alignment requirement for the type of the lvalue *expr*, or 1 if there is no alignment requirement.

Example

```
int Alignment_0(void)
{
    return __alignof__(int);
}
```

See also

- `__ALIGNOF__`.

4.1.3 `__ALIGNOF__`

The `__ALIGNOF__` keyword returns the alignment requirement for a specified type, or for the type of a specified object.

Syntax

`__ALIGNOF__(type)`

`__ALIGNOF__(expr)`

Where:

type is a type

expr is an lvalue.

Return value

`__ALIGNOF__(type)` returns the alignment requirement for the type *type*, or 1 if there is no alignment requirement.

`__ALIGNOF__(expr)` returns the alignment requirement for the type of the lvalue *expr*, or 1 if there is no alignment requirement. The lvalue itself is not evaluated.

Example

```
typedef struct s_foo { int i; short j; } foo;
typedef __packed struct s_bar { int i; short j; } bar;

return __ALIGNOF(struct s_foo); // returns 4
return __ALIGNOF(foo);          // returns 4
return __ALIGNOF(bar);          // returns 1
```

See also

- `__alignof__` on page 4-3.

4.1.4 `__asm`

This keyword is used to pass information from the compiler to the ARM assembler `armasm`.

The precise action of this keyword depends on its usage.

Usage

Embedded assembler

The `__asm` keyword can be used to declare or define an embedded assembly function. For example:

```
__asm void my_strcpy(const char *src, char *dst);
```

See *Embedded assembler* on page 6-17 in the *Compiler User Guide* for more information.

Inline assembler

The `__asm` keyword can be used to incorporate inline assembly into a function. For example:

```
int qadd(int i, int j)
{
    int res;
    __asm
    {
        QADD    res, i, j
    }
    return res;
}
```

See *Inline assembler* on page 6-2 in the *Compiler User Guide* for more information.

Assembler labels

The `__asm` keyword can be used to specify an assembler label for a C symbol. For example:

```
int count __asm__("count_v1"); // export count_v1, not count
```

See *Assembler labels* on page 3-21 for more information.

Named register variables

The `__asm` keyword can be used to declare a named register variable. For example:

```
register int foo __asm("r0");
```

See *Named register variables* on page 4-101 for more information.

See also

- *asm keyword* on page 3-25.

4.1.5 __forceinline

The `__forceinline` keyword forces the compiler to compile a C or C++ function inline.

The semantics of `__forceinline` are exactly the same as those of the C++ `inline` keyword. The compiler attempts to inline a function qualified as `__forceinline`, regardless of its characteristics. However, the compiler does not inline a function if doing so causes problems. For example, a recursive function is inlined into itself only once.

`__forceinline` is a storage class qualifier. It does not affect the type of a function.

———— Note —————

This keyword has the function attribute equivalent `__attribute__((always_inline))`.

Example

```
__forceinline static int max(int x, in y)
{
    return x > y ? x : y; // always inline if possible
}
```

See also

- *--forceinline* on page 2-39
- *__attribute__((always_inline))* on page 4-30.

4.1.6 `__global_reg`

The `__global_reg` storage class specifier allocates the declared variable to a global variable register.

Syntax

```
__global_reg(n) type varName
```

Where:

<i>n</i>	Is an integer between one and eight.
<i>type</i>	Is one of the following types: • any integer type, except long long • any char type • any pointer type.
<i>varName</i>	Is the name of a variable.

Restrictions

If you use this storage class, you cannot use any additional storage class such as **extern**, **static**, or **typedef**.

In C, global register variables cannot be qualified or initialized at declaration. In C++, any initialization is treated as a dynamic initialization.

The number of available registers varies depending on the variant of the AAPCS being used, there are between five and seven registers available for use as global variable registers.

In practice, it is recommended that you do not use more than:

- three global register variables in ARM or Thumb-2
- one global register variable in Thumb-1
- half the number of available floating-point registers as global floating-point register variables.

If you declare too many global variables, code size increases significantly. In some cases, your program might not compile.

Caution

You must take care when using global register variables because:

- There is no check at link time to ensure that direct calls between different compilation units are sensible. If possible, define global register variables used in a program in each compilation unit of the program. In general, it is best to place the definition in a global header file. You 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 function using a global register is called from a compilation unit that does not declare the global register variable, the function reads the wrong values from its supposed global register variables.
 - This storage class can only be used at file scope.
-

Example

Example 4-1 declares a global variable register allocated to r5.

Example 4-1 Declaring a global integer register variable

```
__global_reg(2) int x      ; v2 is the synonym for r5
```

Example 4-2 produces an error because global register must be specified in all declarations of the same variable.

Example 4-2 Global register - declaration error

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

In C, `__global_reg` variables cannot be initialized at definition. Example 4-3 on page 4-9 produces an error in C, but not in C++.

Example 4-3 Global register - initialization error

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

4.1.7 __inline

The `__inline` keyword suggests to the compiler that it compiles a C or C++ function inline, if it is sensible to do so.

The semantics of `__inline` are exactly the same as those of the C++ `inline` keyword.

`__inline` is a storage class qualifier. It does not affect the type of a function.

Example

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

int g(int x, int y)
{
    return f(x) + f(y);
}
```

See also

- *Inlining* on page 4-19 in the *Compiler User Guide*.

4.1.8 __int64

The `__int64` keyword is a synonym for the keyword sequence `long long`.

`__int64` is accepted even when using `--strict`.

See also

- `--[no_]strict` on page 2-84
- `long long` on page 3-8.

4.1.9 __INTADDR__

The `__INTADDR__` operation treats the enclosed expression as a constant expression, and converts it to an integer constant.

Note

This is used in the `offsetof` macro.

Syntax

`__INTADDR(expr)`

Where:

`expr` is an integral constant expression.

Return value

`__INTADDR__(expr)` returns an integer constant equivalent to `expr`.

See also

- *Restrictions on embedded assembly* on page 6-19 in the *Compiler User Guide*.

4.1.10 `__irq`

The `__irq` keyword enables a C or C++ function to be used as an interrupt routine.

`__irq` is a function qualifier. It affects the type of the function.

Restrictions

All corrupted registers except floating-point registers are preserved, not only those that are normally preserved under the AAPCS. The default AAPCS mode must be used.

The function exits by setting the program counter to `1r-4` and the CPSR to the value in SPSR. No arguments or return values can be used with `__irq` functions.

Note

When compiling for Thumb using the `--thumb` option or `#pragma thumb`, any functions specified as `__irq` are compiled for ARM.

See also

- `--thumb` on page 2-86
- `#pragma thumb` on page 4-62
- Chapter 6 *Handling Processor Exceptions* in the *Developer Guide*.

4.1.11 `__packed`

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

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

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.

Usage

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. Only packing fields in a structure that requires packing can reduce the number of unaligned accesses.

Note

On ARM processors that do not support unaligned access in hardware, for example, pre-ARMv6, access to unaligned data can be costly in terms of code size and execution speed. Data accesses through packed structures must be minimized to avoid increase in code size and performance loss.

Restrictions

The following restrictions apply to the use of `__packed`:

- The `__packed` qualifier cannot be used on structures that were previously declared without `__packed`.
- Unlike other type qualifiers you cannot have both a `__packed` and non-`__packed` version of the same structure type.
- The `__packed` qualifier does not affect local variables of integral type.
- A packed structure or union is not assignment-compatible with the corresponding unpacked structure. Because the structures have a different memory layout, the only way to assign a packed structure to an unpacked structure is by a field-by-field copy.

- The effect of casting away `__packed` is undefined. The effect of casting a non packed structure to 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. You can also cast away the `__packed` on `char` types.
- There are no packed array types. A packed array is an array of objects of packed type. There is no padding in the array.

Example

Example 4-4 shows that a pointer can point to a packed type.

Example 4-4 Pointer to packed

```
typedef __packed int* PpI;           /* pointer to a __packed int */
__packed int *p;                    /* pointer to a __packed int */
PpI p2;                             /* 'p2' has the same type as 'p' */
                                   /* __packed is a qualifier */
                                   /* just like 'const' or 'volatile' */

typedef int *PI;                    /* pointer to int */
__packed PI p3;                     /* a __packed pointer to a normal int */
                                   /* -- not the same type as 'p' and 'p2' */

int *__packed p4;                   /* 'p4' has the same type as 'p3' */
```

Example 4-5 shows that when a packed object is accessed using a pointer, the compiler generates code that works and that is independent of the pointer alignment.

Example 4-5 Packed structure

```
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
}

```

See also

- `__attribute__((packed))` on page 4-44
- *Packed structures* on page 5-10
- *The `__packed` qualifier and unaligned accesses to data* on page 4-24 in the *Compiler User Guide*
- *`__packed` structures versus individually `__packed` fields* on page 4-26 in the *Compiler User Guide*.

4.1.12 `__pure`

The `__pure` keyword asserts that a function declaration is pure.

A function is *pure* only if:

- the result depends exclusively on the values of its arguments
- the function has no side effects.

`__pure` is a function qualifier. It affects the type of a function.

Note

This keyword has the function attribute equivalent `__attribute__((const))`.

Default

By default, functions are assumed to be impure.

Usage

Pure functions are candidates for common subexpression elimination.

Restrictions

A function that is declared as pure can have no side effects. For example, pure functions:

- cannot call impure functions
- cannot use global variables or dereference pointers, because the compiler assumes that the function does not access memory, except stack memory
- must return the same value each time when called twice with the same parameters.

Example

```
int factr(int n) __pure
{
    int f = 1;

    while (n > 0)
        f *= n--;
    return f;
}
```

See also

- `__attribute__((const))` on page 4-31
- `__pure` on page 4-15 in the *Compiler User Guide*
- *Placing ARM function qualifiers* on page 4-17 in the *Compiler User Guide*.

4.1.13 `__smc`

The `__smc` keyword declares an SMC (*Secure Monitor Call*) function. A call to the SMC function inserts an SMC instruction into the instruction stream generated by the compiler at the point of function invocation.

————— **Note** —————

The SMC instruction replaces the SMI instruction used in previous versions of the ARM assembly language.

`__smc` is a function qualifier. It affects the type of a function.

Syntax

```
__smc(int smc_num) return-type function-name([argument-list]);
```

Where:

smc_num

Is a 16-bit immediate value used in the SMC instruction.

The value of *smc_num* is ignored by the ARM processor, but can be used by the SMC exception handler to determine what service is being requested.

Restrictions

The SMC instruction is available for selected ARM architecture-based processors, if they have the Security Extensions. See *SMC* on page 4-137 in the *Assembler Guide* for more information.

The compiler generates an error if you compile source code containing the `__smc` keyword for an architecture that does not support the SMC instruction.

Example

```
__smc(5) void mycall(void); /* declare a name by which SMC #5 can be called */

...
mycall();                  /* invoke the function */
```

See also

- *SMC* on page 4-137 in the *Assembler Guide*.

4.1.14 `__softfp`

The `__softfp` keyword asserts that a function uses software floating-point linkage.

`__softfp` is a function qualifier. It affects the type of the function.

————— **Note** —————

This keyword has the `#pragma` equivalent `#pragma __softfp_linkage`.

Usage

Calls to the function pass floating-point arguments in integer registers. If the result is a floating-point value, the value is returned in integer registers. This duplicates the behavior of compilation targeting software floating-point.

This keyword enables the same library to be used by sources compiled to use hardware and software floating-point.

Note

In C++, if a virtual function qualified with the `__softfp` keyword is to be overridden, the overriding function must also be declared as `__softfp`. If the functions do not match, the compiler generates an error.

See also

- `--fpu=name` on page 2-42
- `#pragma [no_]softfp_linkage` on page 4-59
- *Floating-point linkage* on page 4-33 in the *Compiler User Guide*.

4.1.15 `__svc`

The `__svc` keyword declares a *SuperVisor Call* (SVC) function taking up to four integer-like arguments and returning up to four results in a `value_in_regs` structure.

`__svc` is a function qualifier. It affects the type of a function.

Syntax

```
__svc(int svc_num) return-type function-name([argument-list]);
```

Where:

- | | |
|----------------|--|
| <i>svc_num</i> | Is the immediate value used in the SVC instruction.
It is an expression evaluating to an integer in the range: |
| | <ul style="list-style-type: none"> • 0 to $2^{24}-1$ (a 24-bit value) in an ARM instruction • 0-255 (an 8-bit value) in a 16-bit Thumb instruction. |

Usage

This causes function invocations to be compiled inline as an AAPCS-compliant operation that behaves similarly to a normal call to a function.

The `__value_in_regs` qualifier can be used to specify that a small structure of up to 16 bytes is returned in registers, rather than by the usual structure-passing mechanism defined in the AAPCS.

Example

```
__svc(42) void terminate_1(int procnum); // terminate_1 returns no results
__svc(42) int terminate_2(int procnum); // terminate_2 returns one result
```

```
typedef struct res_type
{
    int res_1;
    int res_2;
    int res_3;
    int res_4;
} res_type;

__svc(42) __value_in_regs res_type terminate_3(int procnum);
                                                // terminate_3 returns more than
                                                // one result
```

Errors

When an ARM architecture variant or ARM architecture-based processor that does not support an SVC instruction is specified on the command line using the `--cpu` option, the compiler generates an error.

See also

- `--cpu=name` on page 2-15
- `__value_in_regs` on page 4-19
- *SVC* on page 4-130 in the *Assembler Guide*.

4.1.16 __svc_indirect

The `__svc_indirect` keyword passes an operation code to the SVC handler in r12.

`__svc_indirect` is a function qualifier. It affects the type of a function.

Syntax

```
__svc_indirect(int svc_num)
    return-type function-name(int real_num[, argument-list]);
```

Where:

<i>svc_num</i>	Is the immediate value used in the SVC instruction. It is an expression evaluating to an integer in the range: • 0 to $2^{24}-1$ (a 24-bit value) in an ARM instruction • 0-255 (an 8-bit value) in a 16-bit Thumb instruction.
<i>real_num</i>	Is the value passed in r12 to the handler to determine the function to perform.

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

Usage

You can use this feature to implement indirect SVCs.

Example

```
int __svc_indirect(0) ioctl(int svcino, int fn, void *argp);
```

Calling:

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

compiles to SVC #0 with IOCTL+4 in r12.

Errors

When an ARM architecture variant or ARM architecture-based processor that does not support an SVC instruction is specified on the command line using the `--cpu` option, the compiler generates an error.

See also

- `--cpu=name` on page 2-15
- `__value_in_regs` on page 4-19
- SVC on page 4-130 in the *Assembler Guide*.

4.1.17 __svc_indirect_r7

The `__svc_indirect` keyword behaves like `__svc_indirect`, but uses r7 instead of r12.

`__svc_indirect_r7` is a function qualifier. It affects the type of a function.

Syntax

```
__svc_indirect_r7(int svc_num)
    return-type function-name(int real_num[, argument-list]);
```

Where:

- svc_num* Is the immediate value used in the SVC instruction.
- It is an expression evaluating to an integer in the range:
- 0 to $2^{24}-1$ (a 24-bit value) in an ARM instruction

- 0-255 (an 8-bit value) in a 16-bit Thumb instruction.

real_num Is the value passed in r7 to the handler to determine the function to perform.

Usage

Thumb applications on ARM Linux use `__svc_indirect_r7` to make kernel syscalls.

You can also use this feature to implement indirect SVCs.

Example

```
long __svc_indirect_r7(0) \
    SVC_write(unsigned, int fd, const char * buf, size_t count);
#define write(fd, buf, count) SVC_write(4, (fd), (buf), (count))
```

Calling:

```
write(fd, buf, count);
```

compiles to SVC #0 with r0 = fd, r1 = buf, r2 = count, and r7 = 4.

Errors

When an ARM architecture variant or ARM architecture-based processor that does not support an SVC instruction is specified on the command line using the `--cpu` option, the compiler generates an error.

See also

- `__value_in_regs`
- `--cpu=name` on page 2-15
- SVC on page 4-130 in the *Assembler Guide*.

4.1.18 `__value_in_regs`

The `__value_in_regs` qualifier 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.

`__value_in_regs` is a function qualifier. It affects the type of a function.

Syntax

```
__value_in_regs return-type function-name([argument-list]);
```

Where:

return-type is the type of a structure of up to four words in size.

Usage

Declaring a function `__value_in_regs` can be useful when calling functions that return more than one result.

Restrictions

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

If a virtual function declared as `__value_in_regs` is to be overridden, the overriding function must also be declared as `__value_in_regs`. If the functions do not match, the compiler generates an error.

Errors

Where the structure returned in a function qualified by `__value_in_regs` is too big, a warning is produced and the `__value_in_regs` structure is then ignored.

Example

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

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

See also

- `__value_in_regs` on page 4-14 in the *Compiler User Guide*.

4.1.19 `__weak`

This keyword instructs the compiler to export symbols weakly.

The `__weak` keyword can be applied to function and variable declarations, and to function definitions.

Usage

Functions and variable declarations

For declarations, this storage class specifies an **extern** object declaration that, even if not present, does not cause the linker to fault an unresolved reference.

For example:

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

If the reference to a missing weak function is made from code that compiles to a branch or branch link instruction, then either:

- The reference is resolved as branching to the next instruction. This effectively makes the branch a NOP.
- The branch is replaced by a NOP instruction.

Function definitions

Functions defined with `__weak` export their symbols weakly. A weakly defined function behaves like a normally defined function unless a non weakly defined function of the same name is linked into the same image. If both a non weakly defined function and a weakly defined function exist in the same image then all calls to the function resolve to call the non weak function. If multiple weak definitions are available, the linker chooses one for use by all calls.

Functions declared with `__weak` and then defined without `__weak` behave as non weak functions.

Restrictions

There are restrictions when you qualify function and variable declarations, and function definitions, with `__weak`.

Functions and variable declarations

A function or variable cannot be used both weakly and non weakly in the same compilation. 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 variable weakly from the same compilation that defines the function or variable. The following code uses `f()` non weakly from `h()`:

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

The linker does not load the function or variable from a library unless another compilation uses the function or variable non weakly. If the reference remains unresolved, its value is assumed to be NULL. Unresolved references, however, are not NULL if the reference is from code to a position-independent section or to a missing `__weak` function.

Function definitions

Weakly defined functions cannot be inlined.

Example

```
__weak const int c;           // assume 'c' is not present in final link
const int *f1() { return &c; } // '&c' returns non-NULL if
                               // compiled and linked /ropi

__weak int i;                 // assume 'i' is not present in final link
int *f2() { return &i; }      // '&i' returns non-NULL if
                               // compiled and linked /rwp

__weak void f(void);          // assume 'f' is not present in final link
typedef void (*FP)(void);
FP g() { return f; }          // 'g' returns non-NULL if
                               // compiled and linked /ropi
```

See also

- Chapter 7 *Using the ARM Librarian* in the *Linker Guide* for more information on library searching.

4.1.20 __writeonly

The `__writeonly` type qualifier indicates that a data object cannot be read from.

In the C and C++ type system it behaves as a cv-qualifier like **const** or **volatile**. Its specific effect is that an lvalue with `__writeonly` type cannot be converted to an rvalue.

Assignment to a `__writeonly` bitfield is not allowed if the assignment is implemented as read-modify-write. This is implementation-dependent.

Example

```
void foo(__writeonly int *ptr)
{
    *ptr = 0;                // allowed
    printf("ptr value = %d\n", *ptr); // error
}
```

4.2 __declspec attributes

The `__declspec` keyword enables you to specify special attributes of objects and functions. For example, you can use the `__declspec` keyword to declare imported or exported functions and variables, or to declare *Thread Local Storage* (TLS) objects.

The `__declspec` keyword must prefix the declaration specification. For example:

```
__declspec(noreturn) void overflow(void);
__declspec(thread) int i;
```

Table 4-2 summarizes the available `__declspec` attributes. `__declspec` attributes are storage class modifiers. They do not affect the type of a function or variable.

Table 4-2 __declspec attributes supported by the compiler and their equivalents

__declspec attribute	non __declspec equivalent
__declspec(dllexport)	-
__declspec(dllimport)	-
__declspec(noinline)	__attribute__((noinline))
__declspec(noreturn)	__attribute__((noreturn))
__declspec(nothrow)	-
__declspec(notshared)	-
__declspec(thread)	-

4.2.1 __declspec(dllexport)

The `__declspec(dllexport)` attribute exports the definition of a symbol through the dynamic symbol table when building DLL libraries.

Usage

When an inline function is marked `__declspec(dllexport)`, the function definition might be inlined, but an out-of-line instance of the function is always generated, and exported, in the same way as for a non-inline function.

See also

- `__declspec(dllimport)` on page 4-25.

4.2.2 `__declspec(dllimport)`

The `__declspec(dllimport)` attribute imports a symbol through the dynamic symbol table when building DLL libraries.

Usage

When an inline function is marked `__declspec(dllimport)`, the function definition in this compilation unit might be inlined, but is never generated out-of-line. An out-of-line call or address reference uses the imported symbol.

The `dllimport` attribute can be used on **extern** functions and variables.

See also

- `__declspec(dllexport)` on page 4-24.

4.2.3 `__declspec(noinline)`

The `__declspec(noinline)` attribute suppresses the inlining of a function at the call points of the function.

———— Note ————

This `__declspec` attribute has the function attribute equivalent `__attribute__((noinline))`.

See also

- `#pragma [no_]inline` on page 4-56
- `__attribute__((noinline))` on page 4-32.

4.2.4 `__declspec(noreturn)`

The `__declspec(noreturn)` attribute asserts that a function never returns.

———— Note ————

This `__declspec` attribute has the function attribute equivalent `__attribute__((noreturn))`.

Usage

Use this attribute to reduce the cost of calling a function that never returns, such as `exit()`. If a `noreturn` function returns to its caller, the behavior is undefined.

Restrictions

The return address is not preserved when calling the noreturn function. This limits the ability of a debugger to display the call stack.

Example

```
__declspec(noreturn) void overflow(void); // never return on overflow

int negate(int x)
{
    if (x == 0x80000000) overflow();
    return -x;
}
```

See also

- `__attribute__((noreturn))` on page 4-33.

4.2.5 `__declspec(nothrow)`

The `__declspec(nothrow)` attribute asserts that a call to a function never results in a C++ exception being propagated from the call into the caller.

The ARM library headers automatically add this qualifier to declarations of C functions that, according to the ISO C Standard, can never throw.

Usage

If the compiler knows that a function can never throw out, it might be able to generate smaller exception-handling tables for callers of that function.

Restrictions

If a call to a function results in a C++ exception being propagated from the call into the caller, the behavior is undefined.

This modifier is ignored when not compiling with exceptions enabled.

Example

```
struct S
{
    ~S();
}
```

```

};

__declspec(nothrow) extern void f(void);

void g(void)
{
    S s;
    f();
}

```

See also

- `--[no_]force_new_nothrow` on page 2-38
- *Using the `::operator new` function* on page 5-13.

4.2.6 `__declspec(notshared)`

The `__declspec(notshared)` attribute asserts that specific classes of dynamic symbols are not exported.

Example

```

struct __declspec(notshared) X
{
    virtual int f();
};                                     // do not export this

int X::f()
{
    return 1;
}

struct Y : X
{
    virtual int g();
};                                     // do export this

int Y::g()
{
    return 1;
}

```

4.2.7 `__declspec(thread)`

The `__declspec(thread)` attribute asserts that variables are thread-local and have *thread storage duration*, so that the linker arranges for the storage to be allocated automatically when a thread is created.

Note

The keyword `__thread` is supported as a synonym for `__declspec(thread)`.

Restrictions

File-scope thread-local variables cannot be dynamically initialized.

Example

```
__declspec(thread) int i;  
__thread int j;           // same as __declspec(thread) int j;
```

4.3 Function attributes

The `__attribute__` keyword enables you to specify special attributes of variables or structure fields, functions, and types. The keyword format is either:

```
__attribute__ ((attribute1, attribute2, ...))
__attribute__ ((__attribute1__, __attribute2__, ...))
```

For example:

```
void * Function_Attributes_malloc_0(int b) __attribute__ ((malloc));
static int b __attribute__ ((__unused__));
```

Table 4-3 summarizes the available function attributes.

Table 4-3 Function attributes supported by the compiler and their equivalents

Function attribute	non-attribute equivalent
<code>__attribute__((alias))</code>	-
<code>__attribute__((always_inline))</code>	<code>__forceinline</code>
<code>__attribute__((const))</code>	<code>__pure</code>
<code>__attribute__((deprecated))</code>	-
<code>__attribute__((malloc))</code>	-
<code>__attribute__((noinline))</code>	<code>__declspec(noinline)</code>
<code>__attribute__((noreturn))</code>	<code>__declspec(noreturn)</code>
<code>__attribute__((pure))</code>	-
<code>__attribute__((unused))</code>	-
<code>__attribute__((used))</code>	-
<code>__attribute__((weak))</code>	<code>__weak</code>

4.3.1 `__attribute__((alias))`

This function attribute enables you to specify multiple aliases for functions.

Where a function is defined in the current translation unit, the alias call is replaced by a call to the function, and the alias is emitted alongside the original name. Where a function is not defined in the current translation unit, the alias call is replaced by a call

to the real function. Where a function is defined as **static**, the function name is replaced by the alias name and the function is declared external if the alias name is declared external.

Note

This function attribute is a GNU compiler extension that is supported by the ARM compiler.

Note

Variables names might also be aliased using the corresponding variable attribute `__attribute__((alias))`.

Syntax

```
return-type newname([argument-list]) __attribute__((alias("oldname")));
```

Where:

<i>oldname</i>	is the name of the function to be aliased
<i>newname</i>	is the new name of the aliased function.

Example

```
#include <stdio.h>

void foo(void)
{
    printf("%s\n", __FUNCTION__);
}

void bar(void) __attribute__((alias("foo")));

void gazonk(void)
{
    bar(); // calls foo
}
```

See also

- `__attribute__((alias))` on page 4-40.

4.3.2 `__attribute__((always_inline))`

This function attribute indicates that a function must be inlined.

The compiler attempts to inline the function, regardless of the characteristics of the function. However, the compiler does not inline a function if doing so causes problems. For example, a recursive function is inlined into itself only once.

Note

This function attribute is a GNU compiler extension that is supported by the ARM compiler. It has the keyword equivalent `__forceinline`.

Example

```
static int max(int x, int y) __attribute__((always_inline))
{
    return x > y ? x : y; // always inline if possible
}
```

See also

- `--forceinline` on page 2-39
- `__forceinline` on page 4-6.

4.3.3 `__attribute__((const))`

Many functions examine only the arguments passed to them, and have no effects except for the return value. This is a much stricter class than `__attribute__((pure))`, because a function is not permitted to read global memory. If a function is known to operate only on its arguments then it can be subject to common sub-expression elimination and loop optimizations.

Note

This function attribute is a GNU compiler extension that is supported by the ARM compiler. It has the keyword equivalent `__pure`.

Example

```
int Function_Attributes_const_0(int b) __attribute__((const));

int Function_Attributes_const_2(int b)
{
    int aLocal=0;
    aLocal += Function_Attributes_const_0(b);
    aLocal += Function_Attributes_const_0(b);
    return aLocal;
}
```

In this code `Function_Attributes_const_0` might be called once only, with the result being doubled to obtain the correct return value.

See also

- `__attribute__((pure))` on page 4-33
- `__pure` on page 4-15 in the *Compiler User Guide*.

4.3.4 `__attribute__((deprecated))`

This function attribute indicates that a function exists but the compiler must generate a warning if the deprecated function is used.

———— Note ————

This function attribute is a GNU compiler extension that is supported by the ARM compiler.

Example

```
int Function_Attributes_deprecated_0(int b) __attribute__((deprecated));
```

4.3.5 `__attribute__((malloc))`

This function attribute indicates that the function can be treated like `malloc` and the associated optimizations can be performed.

———— Note ————

This function attribute is a GNU compiler extension that is supported by the ARM compiler.

Example

```
void * Function_Attributes_malloc_0(int b) __attribute__((malloc));
```

4.3.6 `__attribute__((noinline))`

This function attribute suppresses the inlining of a function at the call points of the function.

Note

This function attribute is a GNU compiler extension that is supported by the ARM compiler. It has the `__declspec` equivalent `__declspec(noinline)`.

Example

```
static int b __attribute__ ((noinline));
```

See also

- `#pragma [no_]inline` on page 4-56
- `__declspec(noinline)` on page 4-25.

4.3.7 `__attribute__((noreturn))`

This function attribute informs the compiler that the function does not return. The compiler can then perform optimizations by removing the code that is never reached.

Note

This function attribute is a GNU compiler extension that is supported by the ARM compiler. It has the `__declspec` equivalent `__declspec(noreturn)`.

Example

```
int Function_Attributes_NoReturn_0(void) __attribute__ ((noreturn));
```

See also

- `__declspec(noreturn)` on page 4-25.

4.3.8 `__attribute__((pure))`

Many functions have no effects except to return a value, and that the return value depends only on the parameters and global variables. Functions of this kind can be subject to data flow analysis and might be eliminated.

Note

This function attribute is a GNU compiler extension that is supported by the ARM compiler.

Although related, this function attribute is *not* equivalent to the `__pure` keyword. The function attribute equivalent to `__pure` is `__attribute__((const))`.

Example

```
int Function_Attributes_pure_0(int b) __attribute__((pure));
int Function_Attributes_pure_0(int b)
{
    static int aStatic;
    aStatic++;
    return b++;
}

int Function_Attributes_pure_2(int b)
{
    int aLocal=0;
    aLocal += Function_Attributes_pure_0(b);
    return 0;
}
```

The call to `Function_Attributes_pure_0` in this example might be eliminated because its result is not used.

4.3.9 `__attribute__((section))`

The section function attribute enables you to place code in different sections of the image.

Note

This function attribute is a GNU compiler extension that is supported by the ARM compiler.

Example

In the following example, `Function_Attributes_section_0` is placed into the RO section `new_section` rather than `.text`.

```
void Function_Attributes_section_0 (void)
    __attribute__((section ("new_section")));
void Function_Attributes_section_0 (void)
{
    static int aStatic =0;
    aStatic++;
}
```

In the following example, section function attribute overrides the `#pragma arm` section setting.

```
#pragma arm section code="foo"

int f2()
{
    return 1;
}                                     // into the 'foo' area

__attribute__((section("bar"))) int f3()
{
    return 1;
}                                     // into the 'bar' area

int f4()
{
    return 1;
}                                     // into the 'foo' area

#pragma arm section
```

See also

- `#pragma arm section [section_sort_list]` on page 4-50.

4.3.10 `__attribute__((unused))`

The unused function attribute prevents the compiler from generating warnings if the function is not referenced. This does not change the behavior of the unused function removal process.

————— **Note** —————

This function attribute is a GNU compiler extension that is supported by the ARM compiler.

Example

```
static int Function_Attributes_unused_0(int b) __attribute__((unused));
```

4.3.11 `__attribute__((used))`

This function attribute informs the compiler that a function is to be retained in the object, even if it is unreferenced.

Function marked as used are emitted to a single section, in the order they are declared. You can specify the section functions are placed in using `__attribute__((section))`.

Note

This function attribute is a GNU compiler extension that is supported by the ARM compiler.

Note

Variables might also be marked as used, using the corresponding variable attribute `__attribute__((used))`.

Example

```
int lose_this(int);
int keep_this(int) __attribute__((used)); // retained in object file
int keep_this_too(int) __attribute__((used)); // retained in object file
```

See also

- `__attribute__((section))` on page 4-34.
- `__attribute__((used))` on page 4-47.

4.3.12 `__attribute__((weak))`

Functions defined with `__attribute__((weak))` export their symbols weakly.

Functions declared with `__attribute__((weak))` and then defined without `__attribute__((weak))` behave as *weak* functions. This is not the same behavior as the `__weak` keyword.

Note

This function attribute is a GNU compiler extension that is supported by the ARM compiler.

Example

```
extern int Function_Attributes_weak_0 (int b) __attribute__((weak));
```

See also

- `__weak` on page 4-20.

4.4 Type attributes

The `__attribute__` keyword enables you to specify special attributes of variables or structure fields, functions, and types. The keyword format is either:

```
__attribute__ ((attribute1, attribute2, ...))
__attribute__ ((__attribute1__, __attribute2__, ...))
```

For example:

```
void * Function_Attributes_malloc_0(int b) __attribute__ ((malloc));
static int b __attribute__ ((__unused__));
```

Table 4-4 summarizes the available type attributes.

Table 4-4 Type attributes supported by the compiler and their equivalents

Type attribute	non-attribute equivalent
<code>__attribute__((aligned))</code>	<code>__align</code>
<code>__attribute__((packed))</code>	<code>__packed^a</code>
<code>__attribute__((transparent_union))</code>	-

a. The `__packed` qualifier does not affect type in GNU mode.

4.4.1 `__attribute__((aligned))`

The aligned type attribute specifies a minimum alignment for the type.

Note

This type attribute is a GNU compiler extension that is supported by the ARM compiler.

4.4.2 `__attribute__((packed))`

The packed type attribute specifies that a type must have the smallest possible alignment.

Note

This type attribute is a GNU compiler extension that is supported by the ARM compiler.

Errors

The compiler generates a warning message if you use this attribute in a typedef.

See also

- `__packed` on page 4-11
- *Packed structures* on page 5-10
- *The `__packed` qualifier and unaligned accesses to data* on page 4-24 in the *Compiler User Guide*
- *`__packed` structures versus individually `__packed` fields* on page 4-26 in the *Compiler User Guide*.

4.4.3 `__attribute__((transparent_union))`

The `transparent_union` type attribute enables you to specify a *transparent union type*, that is, a union data type qualified with `__attribute__((transparent_union))__`.

When a function is defined with a parameter having transparent union type, a call to the function with an argument of any type in the union results in the initialization of a union object whose member has the type of the passed argument and whose value is set to the value of the passed argument.

When a union data type is qualified with `__attribute__((transparent_union))`, the transparent union applies to all function parameters with that type.

———— Note ————

This type attribute is a GNU compiler extension that is supported by the ARM compiler.

———— Note ————

Individual function parameters might also be qualified with the corresponding `__attribute__((transparent_union))` variable attribute.

Example

```
typedef union { int i; float f; } U __attribute__((transparent_union));

void foo(U u)
{
    static int s;
    s += u.i;    /* Use the 'int' field */
}
```

```
void caller(void)
{
    foo(1);      /* u.i is set to 1 */
    foo(1.0f);   /* u.f is set to 1.0f */
}
```

Mode

Supported in GNU mode only.

See also

- `__attribute__((transparent_union))` on page 4-45.

4.5 Variable attributes

The `__attribute__` keyword enables you to specify special attributes of variables or structure fields, functions, and types. The keyword format is either:

```
__attribute__ ((attribute1, attribute2, ...))
__attribute__ ((__attribute1__, __attribute2__, ...))
```

For example:

```
void * Function_Attributes_malloc_0(int b) __attribute__ ((malloc));
static int b __attribute__ ((__unused__));
```

Table 4-3 on page 4-29 summarizes the available variable attributes.

Table 4-5 Variable attributes supported by the compiler and their equivalents

Variable attribute	non-attribute equivalent
<code>__attribute__((alias))</code>	-
<code>__attribute__((at(address)))</code>	-
<code>__attribute__((aligned))</code>	-
<code>__attribute__((deprecated))</code>	-
<code>__attribute__((packed))</code>	-
<code>__attribute__((section))</code>	-
<code>__attribute__((transparent_union))</code>	-
<code>__attribute__((unused))</code>	-
<code>__attribute__((used))</code>	-
<code>__attribute__((weak))</code>	<code>__weak</code>
<code>__attribute__((zeroinit))</code>	-

4.5.1 `__attribute__((alias))`

This variable attribute enables you to specify multiple aliases for variables.

Where a variable is defined in the current translation unit, the alias reference is replaced by a reference to the variable, and the alias is emitted alongside the original name. Where a variable is not defined in the current translation unit, the alias reference is

replaced by a reference to the real variable. Where a variable is defined as **static**, the variable name is replaced by the alias name and the variable is declared external if the alias is declared external.

Note

Function names might also be aliased using the corresponding function attribute `__attribute__((alias))`.

Syntax

```
type newname __attribute__((alias("oldname")));
```

Where:

<i>oldname</i>	is the name of the variable to be aliased
<i>newname</i>	is the new name of the aliased variable.

Example

```
#include <stdio.h>

int oldname = 1;

extern int newname __attribute__((alias("oldname"))); // declaration

void foo(void)
{
    printf("newname = %d\n", newname); // prints 1
}
```

See also

- `__attribute__((alias))` on page 4-29.

4.5.2 `__attribute__((at(address)))`

This variable attribute enables you to specify the absolute address of a variable.

The variable is placed in its own section, and the section containing the variable is given an appropriate type by the compiler:

- Read-only variables are placed in a section with type RO.
- Initialized read-write variables are placed in a section with type RW.

In particular, variables explicitly initialized to zero are placed in RW not ZI. Such variables are not candidates for the ZI-to-RW optimization of the compiler.

- Uninitialized variables are placed in a section with type ZI.

———— **Note** —————

This variable attribute is not supported by GNU compilers.

Syntax

```
__attribute__((at(address)))
```

Where:

address is the desired address of the variable.

The keyword `__at` comes immediately after the variable name.

Restrictions

The linker is not always able to place sections produced by the `at` variable attribute.

Errors

The linker gives an error message if it is not possible to place a section at a specified address.

Example

```
const int x1 __attribute__((at(0x10000))) = 10; /* RO */
int x2 __attribute__((at(0x12000))) = 10;      /* RW */
int x3 __attribute__((at(0x14000))) = 0;       /* RW, not ZI */
int x4 __attribute__((at(0x16000)));           /* ZI */
```

See also

- *Using `__at` sections to place sections at a specific address* on page 5-37 in the *Linker Guide*.

4.5.3 `__attribute__((aligned))`

The aligned variable attribute specifies a minimum alignment for the variable or structure field, measured in bytes.

Note

This variable attribute is a GNU compiler extension that is supported by the ARM compiler.

Example

```
int Variable_Attributes_aligned_0 __attribute__((aligned (16)));
/* aligned on 16 byte boundary */
short Variable_Attributes_aligned_1[3] __attribute__((aligned));
/* aligns on 4 byte boundary for ARM */
```

See also

- `__align` on page 4-2.

4.5.4 `__attribute__((deprecated))`

The deprecated variable attribute enables the declaration of a deprecated variable without any warnings or errors being issued by the compiler. However, any access to a deprecated variable creates a warning but still compiles. The warning gives the location where the variable is used and the location where it is defined. This helps you to determine why a particular definition is deprecated.

Note

This variable attribute is a GNU compiler extension that is supported by the ARM compiler.

Example

```
extern int Variable_Attributes_deprecated_0 __attribute__((deprecated));
extern int Variable_Attributes_deprecated_1 __attribute__((deprecated));

void Variable_Attributes_deprecated_2()
{
    Variable_Attributes_deprecated_0=1;
    Variable_Attributes_deprecated_1=2;
}
```

Compiling this example generates two warning messages.

4.5.5 `__attribute__((packed))`

The packed variable attribute specifies that a variable or structure field has the smallest possible alignment. That is, one byte for a variable, and one bit for a field, unless you specify a larger value with the aligned attribute.

Note

This variable attribute is a GNU compiler extension that is supported by the ARM compiler.

Example

```
struct
{
    char a;
    int b __attribute__((packed));
} Variable_Attributes_packed_0;
```

See also

- *Packed structures* on page 5-10
- *The `__packed` qualifier and unaligned accesses to data* on page 4-24 in the *Compiler User Guide*
- *`__packed` structures versus individually `__packed` fields* on page 4-26 in the *Compiler User Guide*.

4.5.6 `__attribute__((section))`

Normally, the ARM compiler places the objects it generates in sections like data and bss. However, you might require additional data sections or you might want a variable to appear in a special section, for example, to map to special hardware. The section attribute specifies that a variable must be placed in a particular data section. If you use the section attribute, read-only variables are placed in RO data sections, read-write variables are placed in RW data sections unless you use the `zero_init` attribute. In this case, the variable is placed in a ZI section.

Note

This variable attribute is a GNU compiler extension that is supported by the ARM compiler.

Example

```

/* in R0 section */
const int descriptor[3] __attribute__((section ("descr"))) = { 1,2,3 };

/* in RW section */
long long rw[10] __attribute__((section ("RW")));

/* in ZI section */
long long altstack[10] __attribute__((section ("STACK"), zero_init));/

```

4.5.7 `__attribute__((transparent_union))`

The `transparent_union` variable attribute, attached to a function parameter which is a union, means that the corresponding argument can have the type of any union member, but the argument is passed as if its type were that of the first union member.

Note

The C specification states that the value returned when a union is written as one type and read back with another is undefined. Therefore, a method of distinguishing which type a `transparent_union` is written in must also be passed as an argument.

Note

This variable attribute is a GNU compiler extension that is supported by the ARM compiler.

Note

You can also use this attribute on a typedef for a union data type. In this case it applies to all function parameters with that type.

Mode

Supported in GNU mode only.

Example

```

typedef union
{
    int myint;
    float myfloat;
} transparent_union_t;

```

```

void Variable_Attributes_transparent_union_0(transparent_union_t aUnion
__attribute__((transparent_union)))
{
    static int aStatic;
    aStatic +=aUnion.myint;
}

void Variable_Attributes_transparent_union_1()
{
    int aLocal =0;
    float bLocal =0;
    Variable_Attributes_transparent_union_0(aLocal);
    Variable_Attributes_transparent_union_0(bLocal);
}

```

See also

- `__attribute__((transparent_union))` on page 4-38.

4.5.8 `__attribute__((unused))`

Normally, the compiler warns if a variable is declared but is never referenced. This attribute informs the compiler that you expect a variable to be unused and tells it not issue a warning if it is not used.

———— Note ————

This variable attribute is a GNU compiler extension that is supported by the ARM compiler.

Example

```

void Variable_Attributes_unused_0()
{
    static int aStatic =0;
    int aUnused __attribute__((unused));
    int bUnused;
    aStatic++;
}

```

In this example, the compiler warns that `bUnused` is declared but never referenced, but does not warn about `aUnused`.

———— Note ————

The GNU compiler does not give any warning.

4.5.9 `__attribute__((used))`

This variable attribute informs the compiler that a variable is to be retained in the object, even if it is unreferenced.

Variables marked as used are emitted to a single section, in the order they are declared. You can specify the section into which variables are placed using `__attribute__((section))`.

Note

This variable attribute is a GNU compiler extension that is supported by the ARM compiler.

Note

Functions might also be marked as used using the corresponding function attribute `__attribute__((used))`.

Usage

You can use `__attribute__((used))` to build tables in the object.

Example

```
static int lose_this = 1;
static int keep_this = 2 __attribute__((used)); // retained in object file
static int keep_this_too = 3 __attribute__((used)); // retained in object file
```

See also

- `__attribute__((section))` on page 4-44
- `__attribute__((used))` on page 4-35.

4.5.10 `__attribute__((weak))`

The declaration of a weak variable is permitted, and acts in a similar way to `__weak`.

- in GNU mode:

```
extern int Variable_Attributes_weak_1 __attribute__((weak));
```
- the equivalent in non-GNU mode is:

```
__weak int Variable_Attributes_weak_compare;
```

Note

The extern qualifier is required in GNU mode. In non-GNU mode the compiler assumes that if the variable is not extern then it is treated like any other non weak variable.

Note

This variable attribute is a GNU compiler extension that is supported by the ARM compiler.

See also

- `__weak` on page 4-20.

4.5.11 `__attribute__((zero_init))`

The section attribute specifies that a variable must be placed in a particular data section. The `zero_init` attribute specifies that a variable with no initializer is placed in a ZI data section. If an initializer is specified, an error is reported.

Note

This variable attribute is a GNU compiler extension that is supported by the ARM compiler.

Example

```
__attribute__((zero_init)) int x;           /* in section ".bss" */
__attribute__((section("mybss"), zero_init)) int y; /* in section "mybss" */
```

See also

- `__attribute__((section))` on page 4-44.

4.6 Pragas

The ARM compiler recognizes a number of ARM-specific pragmas. Table 4-6 summarizes the available pragmas.

———— **Note** ————

Pragmas override related command-line options. For example, `#pragma arm` overrides the command-line option `--thumb`.

Table 4-6 Pragas supported by the compiler

Pragas		
#pragma [no_]anon_unions	#pragma [no_]exceptions_unwind	#pragma Otime
#pragma arm	#pragma hdrstop	#pragma pop
#pragma arm section [section_sort_list]	#pragma import symbol_name	#pragma push
#pragma diag_default tag[, tag,...]	#pragma noline	#pragma [no_]softfp_linkage
#pragma diag_error tag[, tag,...]	#pragma no_pch	#pragma unroll [(n)]
#pragma diag_remark tag[, tag,...]	#pragma Onum	#pragma unroll_completely
#pragma diag_suppress tag[, tag,...]	#pragma once	#pragma thumb
#pragma diag_warning tag[, tag,...]	#pragma Ospace	-

4.6.1 #pragma [no_]anon_unions

This pragma enables or disables support for anonymous structures and unions.

Default

The default is `#pragma no_anon_unions`.

See also

- *Anonymous classes, structures and unions* on page 3-20
- `__attribute__((transparent_union))` on page 4-38.

4.6.2 `#pragma arm`

This pragma switches code generation to the ARM instruction set. It overrides the `--thumb` compiler option.

See also

- `--arm` on page 2-7
- `--thumb` on page 2-86
- `#pragma thumb` on page 4-62.

4.6.3 `#pragma arm section [section_sort_list]`

This pragma specifies a section name to be used for subsequent functions or objects. This includes definitions of anonymous objects the compiler creates for initializations.

———— Note —————

You can use `__attribute__((section(...)))` for functions or variables as an alternative to `#pragma arm section`.

Syntax

`#pragma arm section [section_sort_list]`

Where:

section_sort_list specifies an optional list of section names to be used for subsequent functions or objects.

Usage

Use a scatter-loading description file with the ARM linker to control how to place a named section at a particular address in memory.

Restrictions

This option has no effect on:

- Inline functions and their local static variables.

- Template instantiations and their local static variables.
- Elimination of unused variables and functions. However, using `#pragma arm` section enables the linker to eliminate a function or variable that might otherwise be kept because it is in the same section as a used function or variable.
- The order that definitions are written to the object file.

Example

```
int x1 = 5;                      // in .data (default)
int y1[100];                    // in .bss (default)
int const z1[3] = {1,2,3};      // in .constdata (default)

#pragma arm section rdata = "foo", rodata = "bar"

int x2 = 5;                      // in foo (data part of region)
int y2[100];                    // in .bss
int const z2[3] = {1,2,3};      // in bar
char *s2 = "abc";               // s2 in foo, "abc" in .conststring

#pragma arm section rodata

int x3 = 5;                      // in foo
int y3[100];                    // in .bss
int const z3[3] = {1,2,3};      // in .constdata
char *s3 = "abc";               // s3 in foo, "abc" in .conststring

#pragma arm section code = "foo"
int add1(int x)                  // in foo (code part of region)
{
    return x+1;
}

#pragma arm section code
```

See also

- `__attribute__((section))` on page 4-34
- Chapter 5 *Using Scatter-loading Description Files* in the *Linker Guide*.

4.6.4 `#pragma diag_default tag[,tag,...]`

This pragma returns the severity of the diagnostic messages that have the specified tags to the severities that were in effect before any pragmas were issued.

Syntax

```
#pragma diag_default tag[,tag,...]
```

Where:

`tag[,tag,...]` is a comma-separated list of diagnostic message numbers specifying the messages whose severities are to be changed.

At least one diagnostic message number must be specified.

Example

```
// <stdio.h> not #included deliberately
```

```
#pragma diag_error 223
void hello(void)
{
    printf("Hello ");
}

#pragma diag_default 223
void world(void)
{
    printf("world!\n");
}
```

Compiling this code with the option `--diag_warning=223` generates diagnostic messages to report that the function `printf()` is declared implicitly.

The effect of `#pragma diag_default 223` is to return the severity of diagnostic message 223 to Warning severity, as specified by the `--diag_warning` command-line option.

See also

- `--diag_warning=tag[,tag,...]` on page 2-30
- `#pragma diag_error tag[,tag,...]`
- `#pragma diag_remark tag[,tag,...]` on page 4-53
- `#pragma diag_suppress tag[,tag,...]` on page 4-54
- `#pragma diag_warning tag[, tag, ...]` on page 4-54
- *Controlling the output of diagnostic messages* on page 5-4 in the *Compiler User Guide*.

4.6.5 #pragma diag_error tag[,tag,...]

This pragma sets the diagnostic messages that have the specified tags to Error severity.

Syntax

```
#pragma diag_error tag[,tag,...]
```

Where:

`tag[,tag,...]` is a comma-separated list of diagnostic message numbers specifying the messages whose severities are to be changed.
At least one diagnostic message number must be specified.

See also

- `--diag_error=tag[,tag,...]` on page 2-26
- `#pragma diag_default tag[,tag,...]` on page 4-51
- `#pragma diag_remark tag[,tag,...]`
- `#pragma diag_suppress tag[,tag,...]` on page 4-54
- `#pragma diag_warning tag[,tag,...]` on page 4-54
- *Changing the severity of diagnostic messages* on page 5-5 in the *Compiler User Guide*.

4.6.6 #pragma diag_remark tag[,tag,...]

This pragma sets the diagnostic messages that have the specified tags to Remark severity.

`#pragma diag_remark` behaves analogously to `#pragma diag_errors`, except that the compiler sets the diagnostic messages having the specified tags to Remark severity rather than Error severity.

————— Note —————

Remarks are not displayed by default. Use the `--remarks` compiler option to see remark messages.

Syntax

```
#pragma diag_remark tag[,tag,...]
```

Where:

`tag[,tag,...]` is a comma-separated list of diagnostic message numbers specifying the messages whose severities are to be changed.

See also

- `--diag_remark=tag[,tag,...]` on page 2-27
- `--remarks` on page 2-78
- `#pragma diag_default tag[,tag,...]` on page 4-51
- `#pragma diag_error tag[,tag,...]` on page 4-52
- `#pragma diag_suppress tag[,tag,...]`
- `#pragma diag_warning tag[, tag, ...]`
- *Changing the severity of diagnostic messages* on page 5-5 in the *Compiler User Guide*.

4.6.7 #pragma diag_suppress tag[,tag,...]

This pragma disables all diagnostic messages that have the specified tags.

`#pragma diag_suppress` behaves analogously to `#pragma diag_errors`, except that the compiler suppresses the diagnostic messages having the specified tags rather than setting them to have Error severity.

Syntax

```
#pragma diag_suppress tag[,tag,...]
```

Where:

`tag[, tag, ...]` is a comma-separated list of diagnostic message numbers specifying the messages to be suppressed.

See also

- `--diag_suppress=tag[,tag,...]` on page 2-28
- `#pragma diag_default tag[,tag,...]` on page 4-51
- `#pragma diag_error tag[,tag,...]` on page 4-52
- `#pragma diag_remark tag[,tag,...]` on page 4-53
- `#pragma diag_warning tag[, tag, ...]`
- *Suppressing diagnostic messages* on page 5-6 in the *Compiler User Guide*.

4.6.8 #pragma diag_warning tag[, tag, ...]

This pragma sets the diagnostic messages that have the specified tag(s) to Warning severity.

`#pragma diag_remark` behaves analogously to `#pragma diag_errors`, except that the compiler sets the diagnostic messages having the specified tags to Remark severity rather than Error severity.

Syntax

```
#pragma diag_warning tag[,tag,...]
```

Where:

`tag[,tag,...]` is a comma-separated list of diagnostic message numbers specifying the messages whose severities are to be changed.

See also

- `--diag_warning=tag[,tag,...]` on page 2-30
- `#pragma diag_default tag[,tag,...]` on page 4-51
- `#pragma diag_error tag[,tag,...]` on page 4-52
- `#pragma diag_remark tag[,tag,...]` on page 4-53
- `#pragma diag_suppress tag[,tag,...]` on page 4-54
- *Changing the severity of diagnostic messages* on page 5-5 in the *Compiler User Guide*.

4.6.9 `#pragma [no_]exceptions_unwind`

This pragma enables or disables function unwinding at runtime.

Default

The default is `#pragma exceptions_unwind`.

See also

- `--[no_]exceptions` on page 2-35
- `--[no_]exceptions_unwind` on page 2-35
- *Function unwinding at runtime* on page 5-18.

4.6.10 `#pragma hdrstop`

This pragma enables you to specify where the set of precompilation header files end.

This pragma must appear before the first token that does not belong to a preprocessing directive.

See also

- *Precompiled header files* on page 2-16 in the *Compiler User Guide*.

4.6.11 #pragma import *symbol_name*

This pragma generates an importing reference to *symbol_name*. This is the same as the assembler directive:

```
IMPORT symbol_name
```

Syntax

```
#pragma import symbol_name
```

Where:

symbol_name is a symbol to be imported.

Usage

You can use this pragma to select certain features of the C library, such as the heap implementation or real-time division. If a feature described in this book requires a symbol reference to be imported, the required symbol is specified.

See also

- *Building an application with the C library* on page 2-18 in the *Libraries Guide*.

4.6.12 #pragma [no_]inline

This pragma controls inlining, similar to the `--[no_]inline` command-line option. A function defined under `#pragma no_inline` is not inlined into other functions, and does not have its own calls inlined.

The effect of suppressing inlining into other functions can also be achieved by marking the function as `__declspec(noinline)` or `__attribute__((noinline))`.

Default

The default is `#pragma inline`.

See also

- `--[no_]inline` on page 2-49
- `__declspec(noinline)` on page 4-25

- `__attribute__((noinline))` on page 4-32.

4.6.13 `#pragma no_pch`

This pragma suppresses PCH processing for a given source file.

See also

- `--pch` on page 2-70
- *Precompiled header files* on page 2-16 in the *Compiler User Guide*.

4.6.14 `#pragma Onum`

This pragma changes the optimization level.

Syntax

```
#pragma Onum
```

Where:

num is the new optimization level.
The value of *num* is 0, 1, 2 or 3.

See also

- `-Onum` on page 2-66.

4.6.15 `#pragma once`

This pragma enables the compiler to skip subsequent includes of that header file.

`#pragma once` is accepted for compatibility with other compilers, and enables you to use other forms of header guard coding. However, it is preferable to use `#ifndef` and `#define` coding because this is more portable.

Example

The following example shows the placement of a `#ifndef` guard around the body of the file, with a `#define` of the guard variable after the `#ifndef`.

```
#ifndef FILE_H
#define FILE_H
#pragma once           // optional
```

... body of the header file ...

`#endif`

The `#pragma once` is marked as optional in this example. This is because the compiler recognizes the `#ifndef` header guard coding and skips subsequent includes even if `#pragma once` is absent.

4.6.16 `#pragma Ospace`

This pragma instructs the compiler to perform optimizations to reduce image size at the expense of a possible increase in execution time.

See also

- *#pragma Otime*
- *-Ospace* on page 2-68

4.6.17 `#pragma Otime`

This pragma instructs the compiler to perform optimizations to reduce execution time at the expense of a possible increase in image size.

See also

- *#pragma Ospace*
- *-Otime* on page 2-69.

4.6.18 `#pragma pop`

This pragma restores the previously saved pragma state.

See also

- *#pragma push.*

4.6.19 `#pragma push`

This pragma saves the current pragma state.

See also

- *#pragma pop.*

4.6.20 #pragma [no_]softfp_linkage

This pragma controls software floating-point linkage.

#pragma softfp_linkage asserts that all function declarations up to the next #pragma no_softfp_linkage describe functions that use software floating-point linkage.

Note

This pragma has the keyword equivalent __softfp.

Usage

This pragma can be useful when applied to an entire interface specification, located in the header file, without altering that file.

Default

The default is #pragma no_softfp_linkage.

See also

- `__softfp` on page 4-15
- *Floating-point linkage* on page 4-33 in the *Compiler User Guide*.

4.6.21 #pragma unroll [(n)]

This pragma instructs the compiler to unroll a loop by *n* iterations.

Note

Both vectorized and non vectorized loops can be unrolled using #pragma unroll [(n)]. That is, #pragma unroll [(n)] applies to both --vectorize and --no_vectorize.

Syntax

```
#pragma unroll
```

```
#pragma unroll (n)
```

Where:

n is an optional value indicating the number of iterations to unroll.

Default

If you do not specify a value for *n*, the compiler assumes `#pragma unroll (4)`.

Usage

When compiling at `-O3 -Otime`, the compiler automatically unrolls loops where it is beneficial to do so. You can use this pragma to request that the compiler to unroll a loop that has not been unrolled automatically.

Note

Use this `#pragma` only when you have evidence, for example from `--diag_warning=optimizations`, that the compiler is not unrolling loops optimally by itself.

Restrictions

`#pragma unroll [(n)]` can be used only immediately before a **for** loop, a **while** loop, or a **do ... while** loop.

Example

```
void matrix_multiply(float ** __restrict dest, float ** __restrict src1,
    float ** __restrict src2, unsigned int n)
{
    unsigned int i, j, k;

    for (i = 0; i < n; i++)
    {
        for (k = 0; k < n; k++)
        {
            float sum = 0.0f;
            /* #pragma unroll */
            for(j = 0; j < n; j++)
                sum += src1[i][j] * src2[j][k];
            dest[i][k] = sum;
        }
    }
}
```

In this example, the compiler does not normally complete its loop analysis because `src2` is indexed as `src2[j][k]` but the loops are nested in the opposite order, that is, with *j* inside *k*. When `#pragma unroll` is uncommented in the example, the compiler proceeds to unroll the loop four times.

If the intention is to multiply a matrix that is not a multiple of four in size, for example an $n * n$ matrix, `#pragma unroll (m)` might be used instead, where m is some value such that n is an integral multiple of m .

See also

- `--diag_warning=optimizations` on page 2-31
- `-Onum` on page 2-66
- `-Otime` on page 2-69
- `--[no_]vectorize` on page 2-91
- `#pragma unroll_completely`
- *Optimizing loops* on page 4-4 in the *Compiler User Guide*.

4.6.22 #pragma unroll_completely

This pragma instructs the compiler to completely unroll a loop. It has an effect only if the compiler can determine the number of iterations the loop has.

Note

Both vectorized and non vectorized loops can be unrolled using `#pragma unroll_completely`. That is, `#pragma unroll_completely` applies to both `--no_vectorize` and `--vectorize`.

Usage

When compiling at `-O3 -Otime`, the compiler automatically unrolls loops where it is beneficial to do so. You can use this pragma to request that the compiler completely unroll a loop that has not automatically been unrolled completely.

Note

Use this pragma only when you have evidence, for example from `--diag_warning=optimizations`, that the compiler is not unrolling loops optimally by itself.

Restrictions

`#pragma unroll_completely` can only be used immediately before a **for** loop, a **while** loop, or a **do ... while** loop.

Using `#pragma unroll_completely` on an outer loop can prevent vectorization. On the other hand, using `#pragma unroll_completely` on an inner loop might help in some cases.

See also

- *--diag_warning=optimizations* on page 2-31
- *-Onum* on page 2-66
- *-Otime* on page 2-69
- *--[no_]vectorize* on page 2-91
- *#pragma unroll [(n)]* on page 4-59
- *Optimizing loops* on page 4-4 in the *Compiler User Guide*.

4.6.23 #pragma thumb

This pragma switches code generation to the Thumb instruction set. It overrides the *--arm* compiler option.

If you are compiling code for a pre-Thumb-2 processor and using VFP, *any* function containing floating-point operations is compiled for ARM.

See also

- *--arm* on page 2-7
- *--thumb* on page 2-86
- *#pragma arm* on page 4-50.

4.7 Instruction intrinsics

This section describes instruction intrinsics for realizing ARM machine language instructions from C or C++ code. Table 4-7 summarizes the available intrinsics.

Table 4-7 Instruction intrinsics supported by the ARM compiler

Instruction intrinsics		
<code>__breakpoint</code>	<code>__enable_irq</code>	<code>__return_address</code>
<code>__builtin_clz</code>	<code>__fabs</code>	<code>__ror</code>
<code>__builtin_constant_p</code>	<code>__fabsf</code>	<code>__schedule_barrier</code>
<code>__builtin_expect</code>	<code>__force_stores</code>	<code>__semihost</code>
<code>__builtin_frame_address</code>	<code>__ldrex</code>	<code>__sev</code>
<code>__builtin_return_address</code>	<code>__ldrt</code>	<code>__sqrt</code>
<code>__builtin_popcount</code>	<code>__memory_changed</code>	<code>__sqrtf</code>
<code>__cdp</code>	<code>__nop</code>	<code>__ssat</code>
<code>__clrex</code>	<code>__pld</code>	<code>__strex</code>
<code>__clz</code>	<code>__pli</code>	<code>__strt</code>
<code>__current_pc</code>	<code>__qadd</code>	<code>__swp</code>
<code>__current_sp</code>	<code>__qdbl</code>	<code>__usat</code>
<code>__disable_fiq</code>	<code>__qsub</code>	<code>__wfe</code>
<code>__disable_irq</code>	<code>__rbit</code>	<code>__wfi</code>
<code>__enable_fiq</code>	<code>__rev</code>	<code>__yield</code>

4.7.1 `__breakpoint`

This intrinsic inserts a BKPT instruction into the instruction stream generated by the compiler. It enables you to include a breakpoint instruction in your C or C++ code.

Syntax

```
void __breakpoint(int val)
```

Where:

<code>val</code>	is a compile-time constant integer whose range is:
0 ... 65535	if you are compiling source as ARM code
0 ... 255	if you are compiling source as Thumb code.

Errors

The compiler does not recognize the `__breakpoint` intrinsic when compiling for a target that does not support the BKPT instruction. The compiler generates either a warning or an error in this case.

The undefined instruction trap is taken if a BKPT instruction is executed on an architecture that does not support it.

Example

```
void func(void)
{
    ...
    __breakpoint(0xF02C);
    ...
}
```

See also

- *BKPT* on page 4-129 in the *Assembler Guide*.

4.7.2 `__builtin_clz`

This builtin function inserts a CLZ instruction or an equivalent code sequence into the instruction stream generated by the compiler. It is functionally equivalent to the intrinsic `__clz`, except that the return type is `int` instead of `unsigned char`.

————— Note —————

This builtin function is a GNU compiler extension that is supported by the ARM compiler.

See also

- `__clz` on page 4-69
- *CLZ* on page 4-53 in the *Assembler Guide*.

4.7.3 `__builtin_constant_p`

This builtin function determines if a value is known to be constant at compile-time.

Note

This builtin function is a GNU compiler extension that is supported by the ARM compiler.

Mode

Supported in GNU mode only.

Syntax

```
int __builtin_constant_p(...)
```

Where:

... is the argument whose value is to be tested.

Return value

`__builtin_constant_p` returns the integer 1 if the argument is known to be a compile-time constant and 0 if it is not known to be a compile-time constant.

Restrictions

A return value of 0 does not indicate that the value is not a constant, but that the compiler cannot prove it is a constant with the specified value of the `-O` option.

See also

- `-Onum` on page 2-66.

4.7.4 `__builtin_expect`

This intrinsic provides the compiler with branch prediction information. Calling `__builtin_expect(expr, c)` informs the compiler that it is expected that `expr == c`.

Note

This builtin function is a GNU compiler extension that is supported by the ARM compiler.

Syntax

```
long __builtin_expect(long int expr, long int c)
```

Where:

expr is an integral expression
c is a compile-time constant.

Return value

`__builtin_expect` returns the value of *expr*.

Example

```
if (__builtin_expect(i, 0))
{
    foo();                // we do not expect foo to be called,
}                        // because we expect i to be zero
```

4.7.5 `__builtin_frame_address`

This builtin function returns the frame address of the current function. The *frame address* is usually the address of the first word the function pushes on to the stack.

———— Note ————

This builtin function is a GNU compiler extension that is supported by the ARM compiler.

Mode

Supported in GNU mode only.

Syntax

```
void *__builtin_frame_address(unsigned int val)
```

Where:

val is an unsigned integer evaluating to zero.

Return value

`__builtin_frame_address` returns the address of the frame of the current function.

Example

```
void *frame_address(void)
{
    static void *frm_addr;

    frm_addr = __builtin_frame_address(0); // return the sp
    return frm_addr;
}
```

See also

- `__builtin_return_address`
- `__current_sp` on page 4-70.

4.7.6 `__builtin_return_address`

This builtin function returns the return address of the current function.

———— Note ————

This builtin function is a GNU compiler extension that is supported by the ARM compiler.

Mode

Supported in GNU mode only.

Syntax

```
void *__builtin_return_address(unsigned int val)
```

Where:

`val` is an unsigned integer evaluating to zero.

Return value

`__builtin_return_address` returns the value of the link register that is used in returning from the current function.

Example

```
void *return_address_(void)
{
    static void *rtn_addr;
```

```

    rtn_addr = __builtin_return_address(0); // return the 1r
    return rtn_addr;
}

```

See also

- `__builtin_frame_address` on page 4-66
- `__return_address` on page 4-83.

4.7.7 `__builtin_popcount`

This builtin function returns the population count of a specified value, that is, the number of 1-bits in the value.

Syntax

```
int __builtin_popcount(unsigned int val)
```

Where:

val is an unsigned integer whose population is to be counted.

Return value

`__builtin_popcount` returns the number of bits set to 1 in *val*.

Example

```

void foo(void)
{
    int i = __builtin_popcount(3); // returns 2
    int j = __builtin_popcount(5); // returns 2
    int k = __builtin_popcount(8); // returns 1
}

```

4.7.8 `__cdp`

This intrinsic inserts a CDP or CDP2 instruction into the instruction stream generated by the compiler. It enables you to include coprocessor data operations in your C or C++ code.

Syntax

```
__cdp(unsigned int coproc, unsigned int opcode1, unsigned int opcode2)
```

Where:

<i>coproc</i>	Identifies the coprocessor the instruction is for. <i>coproc</i> must be an integer in the range 0 to 15.
<i>opcode1</i>	Is a coprocessor-specific opcode. Add 0x100 to the opcode to generate a CDP2 instruction.
<i>opcode2</i>	Is a coprocessor-specific opcode.

Usage

The use of these instructions depends on the coprocessor. See your coprocessor documentation for more information.

See also

- *CDP and CDP2* on page 4-121 in the *Assembler Guide*.

4.7.9 __clrex

This intrinsic inserts a CLREX instruction into the instruction stream generated by the compiler. It enables you to include a CLREX instruction in your C or C++ code.

Syntax

```
void __clrex(void)
```

Errors

The compiler does not recognize the __clrex intrinsic when compiling for a target that does not support the CLREX instruction. The compiler generates either a warning or an error in this case.

See also

- *CLREX* on page 4-38 in the *Assembler Guide*.

4.7.10 __clz

This intrinsic inserts a CLZ instruction or an equivalent code sequence into the instruction stream generated by the compiler. It enables you to count the number of leading zeros of a data value in your C or C++ code.

Syntax

```
unsigned char __clz(unsigned int val)
```

Where:

val is an **unsigned int**.

Return value

The `__clz` intrinsic returns the number of leading zeros in *val*.

See also

- `__builtin_clz` on page 4-64
- *CLZ* on page 4-53 in the *Assembler Guide*.

4.7.11 __current_pc

This intrinsic enables you to determine the current value of the program counter at the point in your program where the intrinsic is used.

Syntax

```
unsigned int __current_pc(void)
```

Return value

The `__current_pc` intrinsic returns the current value of the program counter at the point in the program where the intrinsic is used.

See also

- `__current_sp`
- `__return_address` on page 4-83
- *Legacy inline assembler that accesses sp, lr, or pc* on page 6-27 in the *Compiler User Guide*.

4.7.12 __current_sp

This intrinsic returns the value of the stack pointer at the current point in your program.

Syntax

```
unsigned int __current_sp(void)
```

Return value

The `__current_sp` intrinsic returns the current value of the stack pointer at the point in the program where the intrinsic is used.

See also

- `__builtin_frame_address` on page 4-66
- `__current_pc` on page 4-70
- `__return_address` on page 4-83
- *Legacy inline assembler that accesses `sp`, `lr`, or `pc` on page 6-27 in the *Compiler User Guide*.*

4.7.13 `__disable_fiq`

This intrinsic disables FIQ interrupts.

Syntax

```
int __disable_fiq(void)
```

———— Note ————

On ARMv7 and ARMv7-M, the `__disable_fiq` intrinsic has prototype:

```
void __disable_fiq(void)
```

Return value

`__fiq` returns the value the FIQ interrupt mask has in the PSR prior to the disabling of FIQ interrupts.

Restrictions

The `__disable_fiq` intrinsic can only be executed in privileged modes, that is, in non user modes. In User mode this intrinsic does not change the interrupt flags in the CPSR.

Example

```
void foo(void)
{
    int was_masked = __disable_fiq();

    /* ... */
}
```

```

        if (!was_masked)
            __enable_fiq();
    }

```

See also

- `__enable_fiq` on page 4-73.

4.7.14 `__disable_irq`

This intrinsic disables IRQ interrupts.

Syntax

```
int __disable_irq(void)
```

Note

On ARMv7 and ARMv7-M, the `__disable_irq` intrinsic has prototype:

```
void __disable_irq(void)
```

Return value

`__irq` returns the value the IRQ interrupt mask has in the PSR prior to the disabling of IRQ interrupts.

Example

```

void foo(void)
{
    int was_masked = __disable_irq();

    /* ... */

    if (!was_masked)
        __enable_irq();
}

```

Restrictions

The `__disable_irq` intrinsic can only be executed in privileged modes, that is, in non user modes. In User mode this intrinsic does not change the interrupt flags in the CPSR.

See also

- `__enable_irq`.

4.7.15 `__enable_fiq`

This intrinsic enables FIQ interrupts.

Syntax

```
void __enable_fiq(void)
```

Restrictions

The `__enable_fiq` intrinsic can only be executed in privileged modes, that is, in non user modes. In User mode this intrinsic does not change the interrupt flags in the CPSR.

See also

- `__disable_fiq` on page 4-71.

4.7.16 `__enable_irq`

This intrinsic enables IRQ interrupts.

Syntax

```
void __enable_irq(void)
```

Restrictions

The `__enable_irq` intrinsic can only be executed in privileged modes, that is, in non user modes. In User mode this intrinsic does not change the interrupt flags in the CPSR.

See also

- `__disable_irq` on page 4-72.

4.7.17 `__fabs`

This intrinsic inserts a VABS instruction or an equivalent code sequence into the instruction stream generated by the compiler. It enables you to obtain the absolute value of a double-precision floating-point value from within your C or C++ code.

Note

The `__fabs` intrinsic is an analogue of the standard C library function `fabs`. It differs from the standard library function in that a call to `__fabs` is guaranteed to be compiled into a single, inline, machine instruction on an ARM architecture-based processor equipped with a VFP coprocessor.

Syntax

```
double __fabs(double val)
```

Where:

`val` is a double-precision floating-point value.

Return value

The `__fabs` intrinsic returns the absolute value of `val` as a **double**.

See also

- `__fabsf`
- *VABS, VNEG, and VSQRT* on page 5-89 in the *Assembler Guide*.

4.7.18 `__fabsf`

This intrinsic is a single-precision version of the `__fabs` intrinsic. It is functionally equivalent to `__fabs`, except that:

- it takes an argument of type **float** instead of an argument of type **double**
- it returns a **float** value instead of a **double** value.

See also

- `__fabs` on page 4-73
- *V{Q}ABS and V{Q}NEG* on page 5-52 in the *Assembler Guide*.

4.7.19 `__force_stores`

This intrinsic causes all variables that are visible outside the current function, such as variables that have pointers to them passed into or out of the function, to be written back to memory if they have been changed.

This intrinsic also acts as a scheduling barrier.

Syntax

void __force_stores(void)

See also

- `__memory_changed` on page 4-78
- `__schedule_barrier` on page 4-85.

4.7.20 `__ldrex`

This intrinsic inserts an instruction of the form LDREX{size} into the instruction stream generated by the compiler. It enables you to load data from memory in your C or C++ code using an LDREX instruction.

Syntax

unsigned int __ldrex(volatile void *ptr)

Where:

ptr Points to the address of the data to be loaded from memory. To specify the size of the data to be loaded, cast the parameter to an appropriate integral type.

Table 4-8 Access widths supported by the `__ldrex` intrinsic

Instruction	Size of data loaded	C cast
LDREXB	unsigned byte	(char *)
LDREXH	unsigned halfword	(short int *)
LDREX	word	(int *)

Return value

The `__ldrex` intrinsic returns the data loaded from the memory address pointed to by *ptr*.

Errors

The compiler does not recognize the `__ldrex` intrinsic when compiling for a target that does not support the LDREX instruction. The compiler generates either a warning or an error in this case.

The `__ldrex` intrinsic does not support access to doubleword data. The compiler generates an error if you specify an access width that is not supported.

Example

```
int foo(void)
{
    int loc = 0xff;
    return __ldrex((volatile char *)loc);
}
```

Compiling this code with the command-line option `--cpu=6k` produces

```
||foo|| PROC
    MOV     r0,#0xff
    LDREXB  r0,[r0]
    BX      lr
    ENDP
```

See also

- `__strex` on page 4-89
- *LDREX and STREX* on page 4-35 in the *Assembler Guide*.

4.7.21 __ldrt

This intrinsic inserts an assembly language instruction of the form `LDR{size}T` into the instruction stream generated by the compiler. It enables you to load data from memory in your C or C++ code using an LDRT instruction.

Syntax

```
unsigned int __ldrt(const volatile void *ptr)
```

Where:

ptr Points to the address of the data to be loaded from memory. To specify the size of the data to be loaded, cast the parameter to an appropriate integral type.

Table 4-9 Access widths supported by the `__ldrt` intrinsic

Instruction ^a	Size of data loaded	C cast
LDRSBT	signed byte	(signed char *)
LDRBT	unsigned byte	(char *)
LDRSHT	signed half word	(signed short int *)
LDRHT	unsigned halfword	(short int *)
LDRT	word	(int *)

a. Or equivalent.

Return value

The `__ldrt` intrinsic returns the data loaded from the memory address pointed to by *ptr*.

Errors

The compiler does not recognize the `__ldrt` intrinsic when compiling for a target that does not support the LDRT instruction. The compiler generates either a warning or an error in this case.

The `__ldrt` intrinsic does not support access to doubleword data. The compiler generates an error if you specify an access width that is not supported.

Example

```
int foo(void)
{
    int loc = 0xff;
    return __ldrt((const volatile int *)loc);
}
```

Compiling this code with the default options produces:

```
||foo|| PROC
      MOV     r0,#0xff
      LDRBT   r1,[r0],#0
```

```
MOV    r2,#0x100
LDRBT  r0,[r2],#0
ORR     r0,r1,r0,LSL #8
BX      lr
ENDP
```

See also

- `--thumb` on page 2-86
- *LDR and STR (User mode)* on page 4-18 in the *Assembler Guide*.

4.7.22 __memory_changed

This intrinsic causes all variables that are visible outside the current function, such as variables that have pointers to them passed into or out of the function, to be written back to memory if they have been changed, and then to be read back from memory.

This intrinsic also acts as a scheduling barrier.

Syntax

```
void __memory_changed(void)
```

See also

- `__force_stores` on page 4-74
- `__schedule_barrier` on page 4-85.

4.7.23 __nop

This intrinsic inserts a NOP instruction or an equivalent code sequence into the instruction stream generated by the compiler. One NOP instruction is generated for each `__nop` intrinsic in the source.

The compiler does not optimize-away the NOP instructions, except for normal unreachable-code elimination. The `__nop` intrinsic also acts as a barrier for instruction scheduling in the compiler. That is, instructions are not moved from one side of the NOP to the other as a result of optimization.

————— Note —————

You can use the `__schedule_barrier` intrinsic to insert a scheduling barrier without generating a NOP instruction.

Syntax

```
void __nop(void)
```

See also

- `__sev` on page 4-87
- `__schedule_barrier` on page 4-85
- `__wfe` on page 4-94
- `__wfi` on page 4-94
- `__yield` on page 4-95
- *NOP, SEV, WFE, WFI, and YIELD* on page 4-139 in the *Assembler Guide*
- *Generic intrinsics* on page 3-3 in the *Compiler User Guide*.

4.7.24 __pld

This intrinsic inserts a data prefetch, for example PLD, into the instruction stream generated by the compiler. It enables you to signal to the memory system from your C or C++ program that a data load from an address is likely in the near future.

Syntax

```
void __pld(...)
```

Where:

... denotes any number of pointer or integer arguments specifying addresses of memory to prefetch.

Restrictions

If the target architecture does not support data prefetching, this intrinsic has no effect.

Example

```
extern int data1;
extern int data2;

volatile int* interrupt = (volatile int *)0x8000;
volatile int* uart = (volatile int *)0x9000;

void get(void)
{
    __pld(data1, data2);
```

```

while (!*interrupt);

*uart = data1;      // trigger uart as soon as interrupt occurs
*(uart+1) = data2;
}

```

See also

- `__pli`
- *PLD and PLI* on page 4-24 in the *Assembler Guide*.

4.7.25 `__pli`

This intrinsic inserts an instruction prefetch, for example PLI, into the instruction stream generated by the compiler. It enables you to signal to the memory system from your C or C++ program that an instruction load from an address is likely in the near future.

Syntax

```
void __pli(...)
```

Where:

... denotes any number of pointer or integer arguments specifying addresses of instructions to prefetch.

Restrictions

If the target architecture does not support instruction prefetching, this intrinsic has no effect.

See also

- `__pld` on page 4-79
- *PLD and PLI* on page 4-24 in the *Assembler Guide*.

4.7.26 `__qadd`

This intrinsic inserts a QADD instruction or an equivalent code sequence into the instruction stream generated by the compiler. It enables you to obtain the saturating add of two integers from within your C or C++ code.

Syntax

```
int __qadd(int va11, int va12)
```

Where:

va11 is the first summand of the saturating add operation
va12 is the second summand of the saturating add operation.

Return value

The `__qadd` intrinsic returns the saturating add of *va11* and *va12*.

See also

- `__qdbl`
- `__qsub`
- `QADD`, `QSUB`, `QDADD`, and `QDSUB` on page 4-94 in the *Assembler Guide*.

4.7.27 `__qdbl`

This intrinsic inserts instructions equivalent to the saturating add of an integer with itself into the instruction stream generated by the compiler. It enables you to obtain the saturating double of an integer from within your C or C++ code.

Syntax

```
int __qdbl(int va1)
```

Where:

va1 is the data value to be doubled.

Return value

The `__qdbl` intrinsic returns the saturating add of *va1* with itself, or equivalently, `__qadd(va1, va1)`.

See also

- `__qadd` on page 4-80.

4.7.28 `__qsub`

This intrinsic inserts a `QSUB` instruction or an equivalent code sequence into the instruction stream generated by the compiler. It enables you to obtain the saturating subtraction of two integers from within your C or C++ code.

Syntax

```
int __qsub(int va11, int va12)
```

Where:

va11 is the minuend of the saturating subtraction operation
va12 is the subtrahend of the saturating subtraction operation.

Return value

The `__qsub` intrinsic returns the saturating subtraction of *va11* and *va12*.

See also

- `__qadd` on page 4-80
- *QADD, QSUB, QDADD, and QDSUB* on page 4-94 in the *Assembler Guide*.

4.7.29 __rbit

This intrinsic inserts an RBIT instruction into the instruction stream generated by the compiler. It enables you to reverse the bit order in a 32-bit word from within your C or C++ code.

Syntax

```
unsigned int __rbit(unsigned int va1)
```

where:

va1 is the data value whose bit order is to be reversed.

Return value

The `__rbit` intrinsic returns the value obtained from *va1* by reversing its bit order.

See also

- *REV, REV16, REVSH, and RBIT* on page 4-64 in the *Assembler Guide*.

4.7.30 __rev

This intrinsic inserts a REV instruction or an equivalent code sequence into the instruction stream generated by the compiler. It enables you to convert a 32-bit big-endian data value into a little-endian data value, or a 32-bit little-endian data value into big-endian data value from within your C or C++ code.

Note

The compiler introduces REV automatically when it recognizes certain expressions.

Syntax

```
unsigned int __rev(unsigned int val)
```

Where:

val is an **unsigned int**.

Return value

The `__rev` intrinsic returns the value obtained from *val* by reversing its byte order.

See also

- *REV, REV16, REVSH, and RBIT* on page 4-64 in the *Assembler Guide*.

4.7.31 `__return_address`

This intrinsic returns the return address of the current function.

Syntax

```
unsigned int __return_address(void)
```

Return value

The `__return_address` intrinsic returns the value of the link register that is used in returning from the current function.

Restrictions

The `__return_address` intrinsic does *not* affect the ability of the compiler to perform optimizations such as inlining, tail-calling, and code sharing. Where optimizations are made, the value returned by `__return_address` reflects the optimizations performed:

No optimization

When no optimizations are performed, the value returned by `__return_address` from within a function *foo* is the return address of *foo*.

Inline optimization

If a function `foo` is inlined into a function `bar` then the value returned by `__return_address` from within `foo` is the return address of `bar`.

Tail-call optimization

If a function `foo` is tail-called from a function `bar` then the value returned by `__return_address` from within `foo` is the return address of `bar`.

See also

- `__builtin_return_address` on page 4-67
- `__current_pc` on page 4-70
- `__current_sp` on page 4-70
- *Legacy inline assembler that accesses `sp`, `lr`, or `pc` on page 6-27 in the Compiler User Guide.*

4.7.32 `__ror`

This intrinsic inserts a ROR instruction or operand rotation into the instruction stream generated by the compiler. It enables you to rotate a value right by a specified number of places from within your C or C++ code.

———— Note ————

The compiler introduces ROR automatically when it recognizes certain expressions.

Syntax

```
unsigned int __ror(unsigned int val, unsigned int shift)
```

Where:

`val` is the value to be shifted right

`shift` is a constant shift in the range 1-31.

Return value

The `__ror` intrinsic returns the value of `val` rotated right by `shift` number of places.

See also

- *ASR, LSL, LSR, ROR, and RRX on page 4-66 in the Assembler Guide.*

4.7.33 `__schedule_barrier`

This intrinsic creates a sequence point where operations before and operations after the sequence point are not merged by the compiler. A scheduling barrier does not cause memory to be updated. If variables are held in registers they are updated in place, and not written out.

This intrinsic is similar to the `__nop` intrinsic, except that no NOP instruction is generated.

Syntax

```
void schedule_barrier(void)
```

See also

- `__nop` on page 4-78

4.7.34 `__semihost`

This intrinsic inserts an SVC or BKPT instruction into the instruction stream generated by the compiler. It enables you to make semihosting calls from C or C++ that are independent of the target architecture.

Syntax

```
int __semihost(int val, const void *ptr)
```

Where:

<i>val</i>	Is the request code for the semihosting request. See Appendix A <i>Semihosting</i> in the <i>Developer Guide</i> for more information.
<i>ptr</i>	Is a pointer to an argument/result block. See Appendix A <i>Semihosting</i> in the <i>Developer Guide</i> for more information.

Return value

See Appendix A *Semihosting* in the *Developer Guide* for more information on the results of semihosting calls.

Usage

Use this intrinsic from C or C++ to generate the appropriate semihosting call for your target and instruction set:

SVC 0x123456	In ARM state for all architectures.
SVC 0xAB	In Thumb state, excluding ARMv7-M. This behavior is not guaranteed on <i>all</i> debug targets from ARM or from third parties.
BKPT 0xAB	For ARMv7-M, Thumb-2 only.

Restrictions

ARM processors prior to ARMv7 use SVC instructions to make semihosting calls. However, if you are compiling for the Cortex-M3 processor, semihosting is implemented using the BKPT instruction.

Example

```
char buffer[100];
...
void foo(void)
{
    __semlhost(0x01, (const void *)buf); // equivalent in thumb state to
                                         // int __svc(0xAB) my_svc(int, int *);
                                         // result = my_svc(0x1, &buffer);
}
```

Compiling this code with the option `--thumb` generates:

```
||foo|| PROC
...
LDR    r1,|L1.12|
MOVS   r0,#1
SVC    #0xab
...

|L1.12|
...
buffer
%      400
```

See also

- `--cpu=list` on page 2-15
- `--thumb` on page 2-86
- `__svc` on page 4-16

- *BKPT* on page 4-129 in the *Assembler Guide*
- *SVC* on page 4-130 in the *Assembler Guide*
- Appendix A *Semihosting* in the *Developer Guide*.

4.7.35 `__sev`

This intrinsic inserts a SEV instruction into the instruction stream generated by the compiler.

Syntax

```
void __sev(void)
```

Errors

The compiler does not recognize the `__sev` intrinsic when compiling for a target that does not support the SEV instruction. The compiler generates either a warning or an error in this case.

See also

- `__nop` on page 4-78
- `__wfe` on page 4-94
- `__wfi` on page 4-94
- `__yield` on page 4-95
- *NOP, SEV, WFE, WFI, and YIELD* on page 4-139 in the *Assembler Guide*.

4.7.36 `__sqrt`

This intrinsic inserts a VFP VSQRT instruction into the instruction stream generated by the compiler. It enables you to obtain the square root of a double-precision floating-point value from within your C or C++ code.

————— Note —————

The `__sqrt` intrinsic is an analogue of the standard C library function `sqrt`. It differs from the standard library function in that a call to `__sqrt` is guaranteed to be compiled into a single, inline, machine instruction on an ARM architecture-based processor equipped with a VFP coprocessor.

Syntax

```
double __sqrt(double val)
```

Where:

val is a double-precision floating-point value.

Return value

The `__sqrt` intrinsic returns the square root of *val* as a **double**.

Errors

The compiler does not recognize the `__sqrt` intrinsic when compiling for a target that is not equipped with a VFP coprocessor. The compiler generates either a warning or an error in this case.

See also

- `__sqrtf`
- *VABS, VNEG, and VSQRT* on page 5-89 in the *Assembler Guide*.

4.7.37 `__sqrtf`

This intrinsic is a single-precision version of the `__sqrt` intrinsic. It is functionally equivalent to `__sqrt`, except that:

- it takes an argument of type **float** instead of an argument of type **double**
- it returns a **float** value instead of a **double** value.

See also

- `__sqrt` on page 4-87
- *VABS, VNEG, and VSQRT* on page 5-89 in the *Assembler Guide*.

4.7.38 `__ssat`

This intrinsic inserts an SSAT instruction into the instruction stream generated by the compiler. It enables you to saturate a signed value from within your C or C++ code.

Syntax

```
int __ssat(int val, unsigned int sat)
```

Where:

val Is the value to be saturated.

sat Is the bit position to saturate to.

sat must be in the range 1 to 32.

Return value

The __ssat intrinsic returns val saturated to the signed range $-2^{sat-1} \leq x \leq 2^{sat-1} - 1$.

Errors

The compiler does not recognize the __ssat intrinsic when compiling for a target that does not support the SSAT instruction. The compiler generates either a warning or an error in this case.

See also

- __usat on page 4-93
- SSAT and USAT on page 4-96 in the Assembler Guide.

4.7.39 __strex

This intrinsic inserts an instruction of the form STREX{size} into the instruction stream generated by the compiler. It enables you to use a STREX instruction in your C or C++ code to store data to memory.

Syntax

int __strex(unsigned int val, volatile void *ptr)

Where:

- val Is the value to be written to memory.
- ptr Points to the address of the data to be written to in memory. To specify the size of the data to be written, cast the parameter to an appropriate integral type.

Table 4-10 Access widths supported by the __strex intrinsic

Instruction	Size of data loaded	C cast
STREXB	unsigned byte	(char *)
STREXH	unsigned halfword	(short int *)
STREX	word	(int *)

Return value

The `__strex` intrinsic returns:

- 0 if the STREX instruction succeeds
- 1 if the STREX instruction is locked out.

Errors

The compiler does not recognize the `__strex` intrinsic when compiling for a target that does not support the STREX instruction. The compiler generates either a warning or an error in this case.

The `__strex` intrinsic does not support access to doubleword data. The compiler generates an error if you specify an access width that is not supported.

Example

```
int foo(void)
{
    int loc=0xff;

    return(!__strex(0x20, (volatile char *)loc));
}
```

Compiling this code with the command-line option `--cpu=6k` produces

```
||foo|| PROC
    MOV     r0,#0xff
    MOV     r2,#0x20
    STREXB  r1,r2,[r0]
    RSBS    r0,r1,#1
    MOVCC   r0,#0
    BX      lr
ENDP
```

See also

- `__ldrex` on page 4-75
- *LDREX and STREX* on page 4-35 in the *Assembler Guide*.

4.7.40 `__strt`

This intrinsic inserts an assembly language instruction of the form `STR{size}T` into the instruction stream generated by the compiler. It enables you to store data to memory in your C or C++ code using an STRT instruction.

Syntax

```
void __strt(unsigned int val, volatile void *ptr)
```

Where:

val Is the value to be written to memory.

ptr Points to the address of the data to be written to in memory. To specify the size of the data to be written, cast the parameter to an appropriate integral type.

Table 4-11 Access widths supported by the __strt intrinsic

Instruction	Size of data loaded	C cast
STRBT	unsigned byte	(char *)
STRHT	unsigned halfword	(short int *)
STRT	word	(int *)

Errors

The compiler does not recognize the __strt intrinsic when compiling for a target that does not support the STRT instruction. The compiler generates either a warning or an error in this case.

The __strt intrinsic does not support access either to signed data or to doubleword data. The compiler generates an error if you specify an access width that is not supported.

Example

```
void foo(void)
{
    int loc=0xff;
    __strt(0x20, (volatile char *)loc);
}
```

Compiling this code produces:

```
||foo|| PROC
    MOV     r0,#0xff
    MOV     r1,#0x20
    STRBT   r1,[r0],#0
    BX      lr
    ENDP
```

See also

- `--thumb` on page 2-86
- *LDR and STR (User mode)* on page 4-18 in the *Assembler Guide*.

4.7.41 `__swp`

This intrinsic inserts a `SWP{size}` instruction into the instruction stream generated by the compiler. It enables you to swap data between memory locations from your C or C++ code.

———— **Note** ————

The use of `SWP` and `SWPB` is deprecated in ARMv6 and above.

Syntax

`unsigned int __swp(unsigned int val, volatile void *ptr)`

where:

- val* Is the data value to be written to memory.
- ptr* Points to the address of the data to be written to in memory. To specify the size of the data to be written, cast the parameter to an appropriate integral type.

Table 4-12 Access widths supported by the `__swp` intrinsic

Instruction	Size of data loaded	C cast
SWPB	unsigned byte	(<code>char *</code>)
SWP	word	(<code>int *</code>)

Return value

The `__swp` intrinsic returns the data value that previously, is in the memory address pointed to by *ptr*, before this value is overwritten by *val*.

Example

```
int foo(void)
{
    int loc=0xff;
    return(__swp(0x20, (volatile int *)loc));
}
```

Compiling this code produces

```
||foo|| PROC
    MOV     r1, #0xff
    MOV     r0, #0x20
    SWP     r0, r0, [r1]
    BX      lr
    ENDP
```

See also

- *SWP and SWPB* on page 4-39 in the *Assembler Guide*.

4.7.42 __usat

This intrinsic inserts a USAT instruction into the instruction stream generated by the compiler. It enables you to saturate an unsigned value from within your C or C++ code.

Syntax

```
int __usat(unsigned int val, unsigned int sat)
```

Where:

<i>val</i>	Is the value to be saturated.
<i>sat</i>	Is the bit position to saturate to. <i>usat</i> must be in the range 0 to 31.

Return value

The `__usat` intrinsic returns *val* saturated to the unsigned range $0 \leq x \leq 2^{sat-1} - 1$.

Errors

The compiler does not recognize the `__usat` intrinsic when compiling for a target that does not support the USAT instruction. The compiler generates either a warning or an error in this case.

See also

- `__ssat` on page 4-88
- *SSAT and USAT* on page 4-96 in the *Assembler Guide*.

4.7.43 `__wfe`

This intrinsic inserts a WFE instruction into the instruction stream generated by the compiler.

Syntax

```
void __wfe(void)
```

Errors

The compiler does not recognize the `__wfe` intrinsic when compiling for a target that does not support the WFE instruction. The compiler generates either a warning or an error in this case.

See also

- `__wfi`
- `__nop` on page 4-78
- `__sev` on page 4-87
- `__yield` on page 4-95
- *NOP, SEV, WFE, WFI, and YIELD* on page 4-139 in the *Assembler Guide*.

4.7.44 `__wfi`

This intrinsic inserts a WFI instruction into the instruction stream generated by the compiler.

Syntax

```
void __wfi(void)
```

Errors

The compiler does not recognize the `__wfi` intrinsic when compiling for a target that does not support the WFI instruction. The compiler generates either a warning or an error in this case.

See also

- `__yield`
- `__nop` on page 4-78
- `__sev` on page 4-87
- `__wfe` on page 4-94
- *NOP, SEV, WFE, WFI, and YIELD* on page 4-139 in the *Assembler Guide*.

4.7.45 `__yield`

This intrinsic inserts a YIELD instruction into the instruction stream generated by the compiler.

Syntax

```
void __yield(void)
```

Errors

The compiler does not recognize the `__yield` intrinsic when compiling for a target that does not support the YIELD instruction. The compiler generates either a warning or an error in this case.

See also

- `__nop` on page 4-78
- `__sev` on page 4-87
- `__wfe` on page 4-94
- `__wfi` on page 4-94
- *NOP, SEV, WFE, WFI, and YIELD* on page 4-139 in the *Assembler Guide*.

4.7.46 **ARMv6 SIMD intrinsics**

The ARM Architecture v6 Instruction Set Architecture adds over sixty SIMD instructions to ARMv6 for the efficient software implementation of high-performance media applications.

The ARM compiler supports intrinsics that map to the ARMv6 SIMD instructions. These intrinsics are available when compiling your code for an ARMv6 architecture or processor. The following list gives the function prototypes for these intrinsics. The function prototypes given in the list describe the primitive or basic forms of the ARMv6 instructions realized by the intrinsics. To obtain the name of the basic instruction realized by an intrinsic, drop the leading underscores (`__`) from the intrinsic name. For example, the `__qadd16` intrinsic corresponds to an ARMv6 QADD16 instruction.

Note

Each ARMv6 SIMD intrinsic is guaranteed to be compiled into a single, inline, machine instruction for an ARM v6 architecture or processor. However, the compiler might use optimized forms of underlying instructions when it detects opportunities to do so.

The ARMv6 SIMD instructions can set the GE[3:0] bits in the *Application Program Status Register* (APSR). The SIMD instructions might update these flags to indicate the “greater than or equal to” status of each 8/16-bit slice of a SIMD operation.

The ARM compiler treats the GE[3:0] bits as a global variable. To access these bits from within your C or C++ program, either:

- access bits 16-19 of the APSR through a named register variable
- use the `__sel` intrinsic to control a SEL instruction.

```

unsigned int __qadd16(unsigned int, unsigned int)
unsigned int __qadd8(unsigned int, unsigned int)
unsigned int __qasx(unsigned int, unsigned int)
unsigned int __qsax(unsigned int, unsigned int)
unsigned int __qsub16(unsigned int, unsigned int)
unsigned int __qsub8(unsigned int, unsigned int)
unsigned int __sadd16(unsigned int, unsigned int)
unsigned int __sadd8(unsigned int, unsigned int)
unsigned int __sasx(unsigned int, unsigned int)
unsigned int __sel(unsigned int, unsigned int)
unsigned int __shadd16(unsigned int, unsigned int)
unsigned int __shadd8(unsigned int, unsigned int)
unsigned int __shasx(unsigned int, unsigned int)
unsigned int __shsax(unsigned int, unsigned int)
unsigned int __shsub16(unsigned int, unsigned int)
unsigned int __shsub8(unsigned int, unsigned int)
unsigned int __smlad(unsigned int, unsigned int, unsigned int)
unsigned long long __smlald(unsigned int, unsigned int, unsigned long long)
unsigned int __smlsd(unsigned int, unsigned int, unsigned int)
unsigned long long __smlsld(unsigned int, unsigned int, unsigned long long)
unsigned int __smuad(unsigned int, unsigned int)
unsigned int __smusd(unsigned int, unsigned int)
unsigned int __ssat16(unsigned int, unsigned int)
unsigned int __ssax(unsigned int, unsigned int)
unsigned int __ssub16(unsigned int, unsigned int)
unsigned int __ssub8(unsigned int, unsigned int)
unsigned int __sxtab16(unsigned int, unsigned int)
unsigned int __sxtb16(unsigned int, unsigned int)
unsigned int __uadd16(unsigned int, unsigned int)
unsigned int __uadd8(unsigned int, unsigned int)
unsigned int __uasx(unsigned int, unsigned int)
unsigned int __uhadd16(unsigned int, unsigned int)
unsigned int __uhadd8(unsigned int, unsigned int)

```

```
unsigned int __uhasx(unsigned int, unsigned int)
unsigned int __uhsax(unsigned int, unsigned int)
unsigned int __uhsb16(unsigned int, unsigned int)
unsigned int __uhsb8(unsigned int, unsigned int)
unsigned int __uqadd16(unsigned int, unsigned int)
unsigned int __uqadd8(unsigned int, unsigned int)
unsigned int __uqasx(unsigned int, unsigned int)
unsigned int __uqsax(unsigned int, unsigned int)
unsigned int __uqsub16(unsigned int, unsigned int)
unsigned int __uqsub8(unsigned int, unsigned int)
unsigned int __usad8(unsigned int, unsigned int)
unsigned int __usada8(unsigned int, unsigned int, unsigned int)
unsigned int __usax(unsigned int, unsigned int)
unsigned int __usat16(unsigned int, unsigned int)
unsigned int __usub16(unsigned int, unsigned int)
unsigned int __usub8(unsigned int, unsigned int)
unsigned int __uxtab16(unsigned int, unsigned int)
unsigned int __uxtb16(unsigned int, unsigned int)
```

See also

- *Named register variables* on page 4-101
- *Registers* on page 2-6 in the *Assembler Guide*
- *SEL* on page 4-62 in the *Assembler Guide*
- Chapter 5 *NEON and VFP Programming* in the *Assembler Guide*.

4.7.47 ETSI basic operations

RVCT supports for the original ETSI family of basic operations described in the ETSI G.729 recommendation *Coding of speech at 8 kbit/s using conjugate-structure algebraic-code-excited linear prediction (CS-ACELP)*.

To make use of the ETSI basic operations in your own code, include the standard header file `dspfns.h`. The intrinsics supplied in `dspfns.h` are listed in Table 4-13.

Table 4-13 ETSI basic operations supported in RVCT

Intrinsics				
abs_s	L_add_c	L_mult	L_sub_c	norm_l
add	L_deposit_h	L_negate	mac_r	round
div_s	L_deposit_l	L_sat	msu_r	saturate
extract_h	L_mac	L_shl	mult	shl

Table 4-13 ETSI basic operations supported in RVCT (continued)

Intrinsics				
extract_l	L_macNs	L_shr	mult_r	shr
L_abs	L_msu	L_shr_r	negate	shr_r
L_add	L_msuNs	L_sub	norm_s	sub

The header file `dspfns.h` also exposes certain status flags as global variables for use in your C or C++ programs. The status flags exposed by `dspfns.h` are listed in Table 4-14.

Table 4-14 ETSI status flags exposed in RVCT

Status flag	Description
Overflow	Overflow status flag. Generally, saturating functions have a sticky effect on overflow.
Carry	Carry status flag.

Example

```

#include <limits.h>
#include <stdint.h>
#include <dspfns.h>           // include ETSI basic operations

int32_t C_L_add(int32_t a, int32_t b)
{
    int32_t c = a + b;

    if (((a ^ b) & INT_MIN) == 0)
    {
        if ((c ^ a) & INT_MIN)
        {
            c = (a < 0) ? INT_MIN : INT_MAX;
        }
    }

    return c;
}

__asm int32_t asm_L_add(int32_t a, int32_t b)
{
    qadd r0, r0, r1
    bx lr
}
    
```

```

int32_t foo(int32_t a, int32_t b)
{
    int32_t c, d, e, f;

    Overflow = 0;          // set global overflow flag

    c = C_L_add(a, b);     // C saturating add
    d = asm_L_add(a, b);   // assembly language saturating add
    e = __qadd(a, b);      // ARM intrinsic saturating add
    f = L_add(a, b);       // ETSI saturating add

    return Overflow ? -1 : c == d == e == f; // returns 1, unless overflow
}

```

See also

- the header file `dspfns.h` for definitions of the ETSI basic operations as a combination of C code and intrinsics
- *ETSI basic operations* on page 3-6 in the *Compiler User Guide*
- ETSI Recommendation G.191: *Software tools for speech and audio coding standardization*
- *ITU-T Software Tool Library 2005 User's manual*, included as part of ETSI Recommendation G.191
- ETSI Recommendation G723.1 : *Dual rate speech coder for multimedia communications transmitting at 5.3 and 6.3 kbit/s*
- ETSI Recommendation G.729: *Coding of speech at 8 kbit/s using conjugate-structure algebraic-code-excited linear prediction (CS-ACELP)*.

4.7.48 C55x intrinsics

The ARM compiler supports the emulation of selected TI C55x compiler intrinsics.

To make use of the TI C55x intrinsics in your own code, include the standard header file `c55x.h`. The intrinsics supplied in `c55x.h` are listed in Table 4-15.

Table 4-15 TI C55x intrinsics supported in RVCT

Intrinsics			
<code>_abss</code>	<code>_lshrs</code>	<code>_rnd</code>	<code>_smas</code>
<code>_count</code>	<code>_lsadd</code>	<code>_norm</code>	<code>_smpy</code>
<code>_divs</code>	<code>_lsmPY</code>	<code>_round</code>	<code>_sneg</code>
<code>_labss</code>	<code>_lsneg</code>	<code>_roundn</code>	<code>_sround</code>
<code>_lmax</code>	<code>_lssh1</code>	<code>_sadd</code>	<code>_sroundn</code>
<code>_lmin</code>	<code>_lssub</code>	<code>_sh1</code>	<code>_ssh1</code>
<code>_lnorm</code>	<code>_max</code>	<code>_shrs</code>	<code>_ssub</code>
<code>_lsh1</code>	<code>_min</code>	<code>_smac</code>	

Example

```

#include <limits.h>
#include <stdint.h>
#include <c55x.h>           // include TI C55x intrinsics

__asm int32_t asm_lsadd(int32_t a, int32_t b)
{
    qadd r0, r0, r1
    bx lr
}

int32_t foo(int32_t a, int32_t b)
{
    int32_t c, d, e;

    c = asm_lsadd(a, b); // assembly language saturating add
    d = __qadd(a, b);    // ARM intrinsic saturating add
    e = _lsadd(a, b);    // TI C55x saturating add

    return c == d == e; // returns 1
}
    
```

See also

- the header file `c55x.h` for further information on the ARM implementation of the C55x intrinsics
- Publications providing information about TI compiler intrinsics are available from Texas Instruments at <http://www.ti.com>.

4.7.49 Named register variables

The compiler enables you to access registers of an ARM architecture-based processor using named register variables. Named register variables are supported:

- at file scope
- at local scope, but not on a function parameter.

Syntax

register *type var-name* `__asm(reg);`

Where:

- type* is the type of the named register variable.
- Any type of the same size as the register being named can be used in the declaration of a named register variable. The type can be a structure, but note that bitfield layout is sensitive to endianness.
- var-name* is the name of the named register variable.
- reg* is a character string denoting the name of a register on an ARM architecture-based processor.
- Registers available for use with named register variables on ARM architecture-based processors are shown in Table 4-16.

Table 4-16 Named registers available on ARM architecture-based processors

Register	Character string for <code>__asm</code>
CPSR	"cpsr" or "apsr"
SPSR	"spsr"
r0 to r12	"r0" to "r12"

Table 4-16 Named registers available on ARM architecture-based processors

Register	Character string for __asm
r13 or sp	"r13" or "sp"
r14 or lr	"r14" or "lr"
r15 or pc	"r15" or "pc".

On targets with a VFP, the registers of Table 4-17 are also available for use with named register variables.

Table 4-17 Named registers available on targets with a VFP

Register	Character string for __asm
FPSID	"fpsid"
FPSCR	"fpscr"
FPEXC	"fpexc"

Example

```

void foo(void)
{
    register int foo __asm("r0");
}
```

In this example, foo is declared as a named register variable for the register r0.

See also

- *Named register variables* on page 3-9 in the *Compiler User Guide*.

4.8 Compiler predefines

This section documents the predefined macros of the ARM compiler.

4.8.1 Predefined macros

Table 4-18 lists the macro names predefined by the ARM compiler for C and C++. Where the value field is empty, the symbol is only defined.

Table 4-18 Predefined macros

Name	Value	When defined
__arm__	—	Always defined for the ARM compiler, even when you specify the --thumb option. See also __ARMCC_VERSION.
__ARMCC_VERSION	ver	Always defined. It is a decimal number, and is guaranteed to increase between releases. The format is <i>PVtbbb</i> where: • <i>P</i> is the major version • <i>V</i> is the minor version • <i>t</i> is the patch release • <i>bbb</i> is the build number. ———— Note ————— Use this to distinguish between RVCT and other tools that define __arm__.
__APCS_INTERWORK	—	When you specify the --apcs /interwork option or set the CPU architecture to ARMv5T or later.
__APCS_ROPI	—	When you specify the --apcs /ropi option.
__APCS_RWPI	—	When you specify the --apcs /rwp option.
__APCS_FPIC	—	When you specify the --apcs /fpic option.
__ARRAY_OPERATORS	—	In C++ compiler mode, to specify that array new and delete are enabled.
__BASE_FILE__	name	Always defined. Similar to __FILE__, but indicates the primary source file rather than the current one (that is, when the current file is an included file).
__BIG_ENDIAN	—	If compiling for a big-endian target.
_BOOL	—	In C++ compiler mode, to specify that bool is a keyword.
__cplusplus	—	In C++ compiler mode.

Table 4-18 Predefined macros (continued)

Name	Value	When defined
<code>__CC_ARM</code>	1	Always set to 1 for the ARM compiler, even when you specify the <code>--thumb</code> option.
<code>__CHAR_UNSIGNED__</code>	–	In GNU mode. It is defined if and only if <code>char</code> is an unsigned type.
<code>__DATE__</code>	<i>date</i>	Always defined.
<code>__EDG__</code>	–	Always defined.
<code>__EDG_IMPLICIT_USING_STD</code>	–	In C++ mode when you specify the <code>--implicit_using_std</code> option.
<code>__EDG_VERSION__</code>	–	Always set to an integer value that represents the version number of the <i>Edison Design Group</i> (EDG) front-end. For example, version 3.8 is represented as 308. <i>The version number of the EDG front-end does not necessarily match the RVCT or RVDS version number.</i>
<code>__EXCEPTIONS</code>	1	In C++ mode when you specify the <code>--exceptions</code> option.
<code>__FEATURE_SIGNED_CHAR</code>	–	When you specify the <code>--signed_chars</code> option (used by <code>CHAR_MIN</code> and <code>CHAR_MAX</code>).
<code>__FILE__</code>	<i>name</i>	Always defined as a string literal.
<code>__FP_FAST</code>	–	When you specify the <code>--fpmode=fast</code> option.
<code>__FP_FENV_EXCEPTIONS</code>	–	When you specify the <code>--fpmode=ieee_full</code> or <code>--fpmode=ieee_fixed</code> options.
<code>__FP_FENV_ROUNDING</code>	–	When you specify the <code>--fpmode=ieee_full</code> option.
<code>__FP_IEEE</code>	–	When you specify the <code>--fpmode=ieee_full</code> , <code>--fpmode=ieee_fixed</code> , or <code>--fpmode=ieee_no_fenv</code> options.
<code>__FP_INEXACT_EXCEPTION</code>	–	When you specify the <code>--fpmode=ieee_full</code> option.
<code>__GNUC__</code>	<i>ver</i>	When you specify the <code>--gnu</code> option. It is an integer that shows the current major version of the GNU mode being used.
<code>__GNUC_MINOR__</code>	<i>ver</i>	When you specify the <code>--gnu</code> option. It is an integer that shows the current minor version of the GNU mode being used.
<code>__GNUMC__</code>	<i>ver</i>	In GNU mode when you specify the <code>--cpp</code> option. It has the same value as <code>__GNUC__</code> .
<code>__IMPLICIT_INCLUDE</code>	–	When you specify the <code>--implicit_include</code> option.
<code>__INTMAX_TYPE__</code>	–	In GNU mode. It defines the correct underlying type for the <code>intmax_t</code> typedef.
<code>__LINE__</code>	<i>num</i>	Always set. It is the source line number of the line of code containing this macro.

Table 4-18 Predefined macros (continued)

Name	Value	When defined
<code>__MODULE__</code>	<i>mod</i>	Contains the filename part of the value of <code>__FILE__</code> .
<code>__NO_INLINE__</code>	–	When you specify the <code>--no_inline</code> option in GNU mode.
<code>__OPTIMIZE_LEVEL</code>	<i>num</i>	Always set to 2 by default, unless you change the optimization level using the <code>-O<i>num</i></code> option.
<code>__OPTIMIZE_SPACE</code>	–	When you specify the <code>-Ospace</code> option.
<code>__OPTIMIZE_TIME</code>	–	When you specify the <code>-Otime</code> option.
<code>__OPTIMIZE__</code>	–	When <code>-O1</code> or <code>-O2</code> is specified in GNU mode.
<code>__OPTIMIZE_SIZE__</code>	–	When <code>-Ospace</code> is specified in GNU mode.
<code>__PLACEMENT_DELETE</code>	–	In C++ mode to specify that placement delete (that is, an operator delete corresponding to a placement operator new , to be called if the constructor throws an exception) is enabled. This is only relevant when using exceptions.
<code>__PTRDIFF_TYPE__</code>	–	In GNU mode. It defines the correct underlying type for the <code>ptrdiff_t</code> typedef .
<code>__RTTI</code>	–	In C++ mode when RTTI is enabled.
<code>__sizeof_int</code>	4	For <code>sizeof(int)</code> , but available in preprocessor expressions.
<code>__sizeof_long</code>	4	For <code>sizeof(long)</code> , but available in preprocessor expressions.
<code>__sizeof_ptr</code>	4	For <code>sizeof(void *)</code> , but available in preprocessor expressions.
<code>__SIZE_TYPE__</code>	–	In GNU mode. It defines the correct underlying type for the <code>size_t</code> typedef .
<code>__SOFTFP__</code>	–	If compiling to use the software floating-point calling standard and library. Set when you specify the <code>--fpu=softvfp</code> option for ARM or Thumb, or when you specify <code>--fpu=softvfp+vfpv2</code> for Thumb.
<code>__STDC__</code>	–	In all compiler modes.
<code>__STDC_VERSION__</code>	–	Standard version information.
<code>__STRICT_ANSI__</code>	–	When you specify the <code>--strict</code> option.
<code>__TARGET_ARCH_ARM</code>	<i>num</i>	The number of the ARM base architecture of the target CPU irrespective of whether the compiler is compiling for ARM or Thumb.
<code>__TARGET_ARCH_THUMB</code>	<i>num</i>	The number of the Thumb base architecture of the target CPU irrespective of whether the compiler is compiling for ARM or Thumb. The value is defined as zero if the target does not support Thumb.

Table 4-18 Predefined macros (continued)

Name	Value	When defined
__TARGET_ARCH_xx	—	xx represents the target architecture and its value depends on the target architecture. For example, if you specify the compiler options <code>--cpu=4T</code> or <code>--cpu=ARM7TDMI</code> then <code>__TARGET_ARCH_4T</code> is defined.
__TARGET_CPU_xx	—	xx represents the target cpu. The value of xx is derived from the <code>--cpu</code> compiler option, or the default if none is specified. For example, if you specify the compiler option <code>--cpu=ARM7TM</code> then <code>__TARGET_CPU_ARM7TM</code> is defined and no other symbol starting with <code>__TARGET_CPU_</code> is defined. If you specify the target architecture, then <code>__TARGET_CPU_generic</code> is defined. If the processor name contains hyphen (-) characters, these are mapped to an underscore (_). For example, <code>--cpu=ARM1136JF-S</code> is mapped to <code>__TARGET_CPU_ARM1136JF_S</code> .
__TARGET_FEATURE_DOUBLEWORD	—	ARMv5T and above.
__TARGET_FEATURE_DSPMUL	—	If the DSP-enhanced multiplier is available, for example ARMv5TE.
__TARGET_FEATURE_MULTIPLY	—	If the target architecture supports the long multiply instructions MULL and MULAL.
__TARGET_FEATURE_DIVIDE	—	If the target architecture supports the hardware divide instruction (that is, ARMv7-M or ARMv7-R).
__TARGET_FEATURE_THUMB	—	If the target architecture supports Thumb, ARMv4T or later.
__TARGET_FPU_xx	—	One of the following is set to indicate the FPU usage: __TARGET_FPU_NONE __TARGET_FPU_VFP __TARGET_FPU_SOFTVFP In addition, if compiling <code>--fpu=softvfp+vfpv2</code> or <code>--fpu=softvfp+vfpv3</code> , <code>__TARGET_FPU_SOFTVFP_VFPV2</code> or <code>__TARGET_FPU_SOFTVFP_VFPV3</code> is also set. See <code>--fpu=name</code> on page 2-42 for more information.
__thumb__	—	When the compiler is in Thumb mode. That is, you have either specified the <code>--thumb</code> option on the command-line or <code>#pragma thumb</code> in your source code. Note - The compiler might generate some ARM code even if it is compiling for Thumb. <code>__thumb</code> and <code>__thumb__</code> become defined or undefined when using <code>#pragma thumb</code> or <code>#pragma arm</code>, but do not change in cases where Thumb functions are generated as ARM code for other reasons (for example, a function specified as <code>__irq</code>).

Table 4-18 Predefined macros (continued)

Name	Value	When defined
__TIME__	<i>time</i>	Always defined.
__UINMAX_TYPE__	–	In GNU mode. It defines the correct underlying type for the <code>uintmax_t</code> typedef.
__VERSION__	<i>ver</i>	When you specify the <code>--gnu</code> option. It is a string that shows the current version of the GNU mode being used.
_WCHAR_T	–	In C++ mode, to specify that <code>wchar_t</code> is a keyword.
__WCHAR_TYPE__	–	In GNU mode. It defines the correct underlying type for the <code>wchar_t</code> typedef.
__WCHAR_UNSIGNED__	–	In GNU mode when you specify the <code>--cpp</code> option. It is defined if and only if <code>wchar_t</code> is an unsigned type.
__WINT_TYPE__	–	In GNU mode. It defines the correct underlying type for the <code>wint_t</code> typedef.

4.8.2 Function names

Table 4-19 lists builtin variables supported by the ARM compiler for C and C++.

Table 4-19 Builtin variables

Name	Value
__FUNCTION__	Holds the name of the function as it appears in the source. __FUNCTION__ is a constant string literal. You cannot use the preprocessor to join the contents to other text to form new tokens.
__PRETTY_FUNCTION__	Holds the name of the function as it appears pretty printed in a language-specific fashion. __PRETTY_FUNCTION__ is a constant string literal. You cannot use the preprocessor to join the contents to other text to form new tokens.

Chapter 5

C and C++ Implementation Details

This chapter describes the language implementation details for the ARM compiler. It includes:

- *C and C++ implementation details* on page 5-2
- *C++ implementation details* on page 5-13.

5.1 C and C++ implementation details

This section describes language implementation details common to both C and C++.

5.1.1 Character sets and identifiers

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

- 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. To permit dollar signs in identifiers with the `--strict` option, also use the `--dollar` command-line option.
- 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 using C macros* on page 2-60 in the *Libraries Guide*)
- Source files are compiled according to the currently selected locale. You might have to select a different locale, with the `--locale` command-line option, if the source file contains non-ASCII characters. See *Invoking the ARM compiler* on page 2-2 in the *Compiler User Guide* for more information.
- The ARM compiler supports multibyte character sets, such as 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.

- Table 5-1 lists all integer character constants, that contain a single character or character escape sequence, are represented in both the source and execution character sets.

Table 5-1 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

- 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 `--signed_chars` option can be used to make the **char** signed
 - the `--unsigned_chars` option can be used to make the **char** unsigned.

———— **Note** —————

Care must be taken when mixing translation units that have been compiled with and without the `--[un]signed_chars` option, and that share interfaces or data structures.

The ARM ABI defines **char** as an unsigned byte, and this is the interpretation used by the C++ libraries supplied with RVCT.

- No locale is used to convert multibyte characters into the corresponding wide characters for a wide character constant. This is not relevant to the generic implementation.

5.1.2 Basic data types

This section describes how the basic data types are implemented in ARM C and C++.

Size and alignment of basic data types

Table 5-2 gives the size and natural alignment of the basic data types.

Table 5-2 Size and alignment of data types

Type	Size in bits	Natural alignment in bytes
char	8	1 (byte-aligned)
short	16	2 (halfword-aligned)
int	32	4 (word-aligned)
long	32	4 (word-aligned)
long long	64	8 (doubleword-aligned)
float	32	4 (word-aligned)
double	64	8 (doubleword-aligned)
long double	64	8 (doubleword-aligned)
All pointers	32	4 (word-aligned)
bool (C++ only)	8	1 (byte-aligned)
_Bool (C only ^a)	8	1 (byte-aligned)
wchar_t (C++ only)	16	2 (halfword-aligned)

a. stdbool.h can be used to define the bool macro in C.

Type alignment varies according to the context:

- Local variables are usually kept in registers, but when local variables spill 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.

See *Structures, unions, enumerations, and bitfields* on page 5-7 for more information.

Integer

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.

Float

Floating-point quantities are stored in IEEE format:

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

For **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. See *Operations on floating-point types* on page 5-6 for more information.

Arrays and pointers

The following statements apply to all pointers to objects in C and C++, except pointers to members:

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

5.1.3 Operations on basic data types

The ARM compiler performs the usual arithmetic conversions set out in relevant sections of the ISO C99 and ISO C++ standards. The following subsections describe additional points that relate to arithmetic operations.

See also *Expression evaluation* on page B-7.

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.
- For values of type `int`,
 - Shifts outside the range 0 to 127 are undefined.
 - 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.
- For values of type `long long`, shifts outside the range 0 to 63 are undefined.
- The remainder on integer division has the same sign as the divisor, as mandated by the ISO C99 standard.
- 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 is too large, positive or negative, for the new type, there is no guarantee that the sign of the result is going to 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 returns zero by default.

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.
- Floating-point exceptions are disabled by default.

Also, see `--fpmode=model` on page 2-39.

————— Note —————

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 floating-point error handling by tailoring the functions and definitions in `fenv.h`. See *Tailoring error signaling, error handling, and program exit* on page 2-76 for more information.

Pointer subtraction

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

- When one pointer is subtracted from another, the difference is the result of the expression:

$$((\text{int})a - (\text{int})b) / (\text{int})\text{sizeof}(\text{type pointed to})$$
- If the pointers point to objects whose alignment is the same as their size, this alignment ensures that division is exact.
- If the pointers point to objects whose alignment is less than their size, such as packed types and most **structs**, both pointers must point to elements of the same array.

5.1.4 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.

See *Anonymous classes, structures and unions* on page 3-20 for more information.

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 storage type of an **enum** is the first of the following, according to the range of the enumerators in the **enum**:

- **unsigned char** if not using `--enum_is_int`
- **signed char** if not using `--enum_is_int`
- **unsigned short** if not using `--enum_is_int`
- **signed short** if not using `--enum_is_int`
- **signed int**
- **unsigned int** except C with `--strict`
- **signed long long** except C with `--strict`
- **unsigned long long** except C with `--strict`.

Implementing **enum** in this way can reduce data size. The command-line option `--enum_is_int` forces the underlying type of **enum** to at least as wide as **int**.

See the description of C language mappings in the *Procedure Call Standard for the ARM Architecture* specification for more information.

Note

Care must be taken when mixing translation units that have been compiled with and without the `--enum_is_int` option, and that share interfaces or data structures.

Handling values that are out of range

In strict C, enumerator values must be representable as **ints**, for example, they must be in the range -2147483648 to +2147483647, inclusive. In previous releases of RVCT out-of-range values were cast to **int** without a warning (unless you specified the `--strict` option).

In RVCT v2.2 and above, a Warning is issued for out-of-range enumerator values:

```
#66: enumeration value is out of "int" range
```

Such values are treated the same way as in C++, that is, they are treated as **unsigned int**, **long long**, or **unsigned long long**.

To ensure that out-of-range Warnings are reported, use the following command to change them into Errors:

```
armcc --diag_error=66 ...
```

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.

- In RVCT v2.0 and above, **double** and **long long** data types are eight-byte aligned. This enables efficient use of the LDRD and STRD instructions in ARMv5TE and above.
- Bitfield alignment depends on how the bitfield is declared. See *Bitfields in packed structures* on page 5-12 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 5-1 shows an example of a conventional, non-packed structure. Bytes 1, 2, and 3 are padded to ensure correct field alignment. Bytes 11 and 12 are padded to ensure correct structure alignment. The `sizeof()` function returns the size of the structure including padding.

```
struct {char c; int x; short s} ex1;
```

0	1	2	3
c	padding		
4	5	7	8
x			
9	10	11	12
s		padding	

Figure 5-1 Conventional non-packed structure example

The compiler pads structures in one of the following 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 with whatever is previously stored in those memory locations. You cannot use `memcmp()` to compare padded structures defined in this way (see Figure 5-1).

Use the `--remarks` option to view the messages that are generated when the compiler inserts padding in a **struct**.

Structures with empty initializers are permitted in C++:

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

However, if you are compiling C, or compiling C++ with the `-cpp` and `--c90` options, an error is generated.

Packed structures

A packed structure is one where the alignment of the structure, and of the fields within it, is always 1.

Packed structures are defined with the `__packed` qualifier. There is no command-line option to change the default packing of structures.

Bitfields

In non-packed structures, the ARM compiler allocates 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 ————

In strict 1990 ISO Standard C, the only types permitted for a bit field are **int**, **signed int**, and **unsigned int**. For non **int** bitfields, the compiler displays an error.

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:

```
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 example structure, 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. Figure 5-2 shows the bitfield allocation for the following example structure:

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

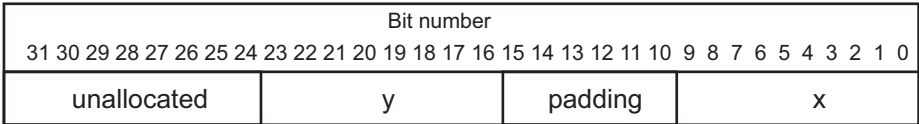


Figure 5-2 Bitfield allocation 1

———— **Note** ————

The same basic rules apply to bitfield declarations with different container types. For example, adding an **int** bitfield to the example structure 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, see Figure 5-3.

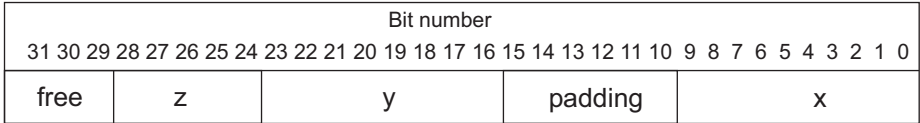


Figure 5-3 Bitfield allocation 2

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 not empty. A subsequent bitfield declaration starts 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 $8 * \text{sizeof}(\text{container-type}) - 1$ bits.

5.2 C++ implementation details

This section describes language implementation details specific to C++.

5.2.1 Using the `::operator new` function

In accordance with the ISO C++ Standard, the `::operator new(std::size_t)` throws an exception when memory allocation fails rather than raising a signal. If the exception is not caught, `std::terminate()` is called.

The compiler option `--force_new_nothrow` turns all new calls in a compilation into calls to `::operator new(std::size_t, std::nothrow_t&)` or `::operator new[](std::size_t, std::nothrow_t&)`. However, this does not affect `operator new` calls in libraries, nor calls to any class-specific `operator new`. See `--[no_]force_new_nothrow` on page 2-38 for more information.

Legacy support

In RVCT v2.0, when the `::operator new` function ran out of memory, it raised the signal **SIGOUTOFHEAP**, instead of throwing a C++ exception. See *ISO C library implementation definition* on page 2-109 in the *Libraries Guide*.

In the current release, it is possible to install a `new_handler` to raise a signal and so restore the RVCT v2.0 behavior.

———— Note ————

Do not rely on the implementation details of this behavior, because it might change in future releases.

5.2.2 Tentative arrays

The ADS v1.2 and RVCT v1.2 C++ compilers enabled you to use tentative, that is, incomplete array declarations, for example, `int a[]`. You cannot use tentative arrays when compiling C++ with the RVCT v2.x compilers or above.

5.2.3 Old-style C parameters in C++ functions

The ADS v1.2 and RVCT v1.2 C++ compilers enabled you to use old-style C parameters in C++ functions. That is,

```
void f(x) int x; { }
```

In the RVCT v2.x compilers or above, you must use the `--anachronisms` compiler option if your code contains any old-style parameters in functions. The compiler warns you if it finds any instances.

5.2.4 Anachronisms

The following anachronisms are accepted when you enable anachronisms using the `--anachronisms` option:

- **overload** is permitted in function declarations. It is accepted and ignored.
- Definitions are not required for static data members that can be initialized using default initialization. The anachronism does not apply to static data members of template classes, because these must always be defined.
- The number of elements in an array can be specified in an array delete operation. The value is ignored.
- A single `operator++()` and `operator--()` function can be used to overload both prefix and postfix operations.
- The base class name can be omitted in a base class initializer if there is only one immediate base class.
- Assignment to the `this` pointer in constructors and destructors is permitted.
- A bound function pointer, that is, a pointer to a member function for a given object, can be cast to a pointer to a function.
- A nested class name can be used as a non-nested class name provided no other class of that name has been declared. The anachronism is not applied to template classes.
- A reference to a non-const type can be initialized from a value of a different type. A temporary is created, it is initialized from the converted initial value, and the reference is set to the temporary.
- A reference to a non const class type can be initialized from an rvalue of the class type or a class derived from that class type. No, additional, temporary is used.
- A function with old-style parameter declarations is permitted and can participate in function overloading as if it were prototyped. Default argument promotion is not applied to parameter types of such functions when the check for compatibility is done, so that the following declares the overloading of two functions named `f`:

```
int f(int);
int f(x) char x; { return x; }
```

Note

In C, this code is legal but has a different meaning. A tentative declaration of `f` is followed by its definition.

5.2.5 Template instantiation

The ARM compiler does all template instantiations automatically, and makes sure there is only one definition of each template entity left after linking. The compiler does this by emitting template entities in named common sections. Therefore, all duplicate common sections, that is, common sections with the same name, are eliminated by the linker.

Note

You can limit the number of concurrent instantiations of a given template with the `--pending_instantiations` compiler option.

See also `--pending_instantiations=n` on page 2-73 for more information.

Implicit inclusion

When implicit inclusion is enabled, the compiler assumes that if it requires a definition to instantiate a template entity declared in a `.h` file it can implicitly include the corresponding `.cc` file to get the source code for the definition. For example, if a template entity `ABC::f` is declared in file `xyz.h`, and an instantiation of `ABC::f` is required in a compilation but no definition of `ABC::f` appears in the source code processed by the compilation, then the compiler checks to see if a file `xyz.cc` exists. If this file exists, the compiler processes the file as if it were included at the end of the main source file.

To find the template definition file for a given template entity the compiler has to know the full path name of the file where the template is declared and whether the file is included using the system include syntax, for example, `#include <file.h>`. This information is not available for preprocessed source containing `#line` directives. Consequently, the compiler does not attempt implicit inclusion for source code containing `#line` directives.

The compiler looks for the definition-file suffixes `.cc` and `.CC`.

You can turn implicit inclusion mode on or off with the command-line options `--implicit_include` and `--no_implicit_include`.

Implicit inclusions are only performed during the normal compilation of a file, that is, when not using the `-E` command-line option.

See *Command-line options* on page 2-2 for more information.

5.2.6 Namespaces

When doing name lookup in a template instantiation, some names must be found in the context of the template definition. Other names can be found in the context of the template instantiation. The compiler implements two different instantiation lookup algorithms:

- the algorithm mandated by the standard, and referred to as dependent name lookup.
- the algorithm that exists before dependent name lookup is implemented.

Dependent name lookup is done in strict mode, unless explicitly disabled by another command-line option, or when dependent name processing is enabled by either a configuration flag or a command-line option.

Dependent name lookup processing

When doing dependent name lookup, the compiler implements the instantiation name lookup rules specified in the standard. This processing requires that non class prototype instantiations be done. This in turn requires that the code be written using the typename and template keywords as required by the standard.

Lookup using the referencing context

When not using dependent name lookup, the compiler uses a name lookup algorithm that approximates the two-phase lookup rule of the standard, but in a way that is more compatible with existing code and existing compilers.

When a name is looked up as part of a template instantiation, but is not found in the local context of the instantiation, it is looked up in a synthesized instantiation context. This synthesized instantiation context includes both names from the context of the template definition and names from the context of the instantiation. For example:

```
namespace N
{
    int g(int);
    int x = 0;
    template <class T> struct A
    {
        T f(T t) { return g(t); }
        T f() { return x; }
    };
}
```

```

namespace M {
    int x = 99;
    double g(double);
    N::A<int> ai;
    int i = ai.f(0);           // N::A<int>::f(int) calls N::g(int)
    int i2 = ai.f();           // N::A<int>::f() returns 0 (= N::x)
    N::A<double> ad;
    double d = ad.f(0);        // N::A<double>::f(double) calls M::g(double)
    double d2 = ad.f();        // N::A<double>::f() also returns 0 (= N::x)
}

```

The lookup of names in template instantiations does not conform to the rules in the standard in the following respects:

- Although only names from the template definition context are considered for names that are not functions, the lookup is not limited to those names visible at the point where the template is defined.
- Functions from the context where the template is referenced are considered for all function calls in the template. Functions from the referencing context are only visible for dependent function calls.

Argument-dependent lookup

When argument-dependent lookup is enabled, functions that are made visible using argument-dependent lookup can overload with those made visible by normal lookup. The standard requires that this overloading occur even when the name found by normal lookup is a block extern declaration. The compiler does this overloading, but in default mode, argument-dependent lookup is suppressed when the normal lookup finds a block extern.

This means a program can have different behavior, depending on whether it is compiled with or without argument-dependent lookup, even if the program makes no use of namespaces. For example:

```

struct A { };
A operator+(A, double);
void f()
{
    A a1;
    A operator+(A, int);
    a1 + 1.0;           // calls operator+(A, double) with arg-dependent lookup
}                      // enabled but otherwise calls operator+(A, int);

```

5.2.7 C++ exception handling

C++ exception handling is fully supported in RVCT. However, the compiler does not support this by default. You must enable C++ exception handling with the `--exceptions` option. See *--[no_]exceptions* on page 2-35 for more information.

Note

The Rogue Wave Standard C++ Library is provided with C++ exceptions enabled.

You can exercise limited control over exception table generation.

Function unwinding at runtime

By default, functions compiled with `--exceptions` can be unwound at runtime. See *--[no_]exceptions* on page 2-35 for more information. *Function unwinding* includes destroying C++ automatic variables, and restoring register values saved in the stack frame. Function unwinding is implemented by emitting an exception table describing the operations to be performed.

You can enable or disable unwinding for specific functions with the pragmas `#pragma exceptions_unwind` and `#pragma no_exceptions_unwind`, see *Pragmas* on page 4-49 for more information. The `--exceptions_unwind` option sets the initial value of this pragma.

Disabling function unwinding for a function has the following effects:

- Exceptions cannot be thrown through that function at runtime, and no stack unwinding occurs for that throw. If the throwing language is C++, then `std::terminate` is called.
- A very compact exception table representation can be used to describe the function, which assists smart linkers with table optimization.
- Function inlining is restricted, because the caller and callee must interact correctly.

Therefore, `#pragma no_exceptions_unwind` can be used to forcibly prevent unwinding in a way that requires no additional source decoration.

By contrast, in C++ an empty function exception specification permits unwinding as far as the protected function, then calls `std::unexpected()` in accordance with the ISO C++ Standard.

5.2.8 Extern inline functions

The ISO C++ Standard requires inline functions to be defined wherever you use them. To prevent the clashing of multiple out-of-line copies of inline functions, the C++ compiler emits out-of-line extern functions in common sections.

Out-of-line inline functions

The compiler emits inline functions out-of-line, in the following cases:

- The address of the function is taken, for example:

```
inline int g()
{
    return 1;
}
int (*fp)() = &g;
```
- The function cannot be inlined, for example, a recursive function:

```
inline int g()
{
    return g();
}
```
- The heuristic used by the compiler decides that it is better not to inline the function. This heuristic is influenced by `-Ospace` and `-Otime`. If you use `-Otime`, the compiler inlines more functions. You can override this heuristic by declaring a function with `__forceinline`. For example:

```
__forceinline int g()
{
    return 1;
}
```

See also `--forceinline` on page 2-39 for more information.

Appendix A

Via File Syntax

This appendix describes the syntax of via files accepted by all the ARM development tools. It contains the following sections:

- *Overview of via files* on page A-2
- *Syntax* on page A-3.

A.1 Overview of via files

Via files are plain text files that contain command-line arguments and options to ARM development tools. You can use via files with all the ARM command-line tools, that is, you can specify a via file from the command line using the `--via` command-line option with:

- `armcc`
- `armasm`
- `armlink`
- `fromelf`
- `armar.`

See the documentation for the individual tool for more information.

Note

In general, you can use a via file to specify any command-line option to a tool, including `--via`. This means that you can call multiple nested via files from within a via file.

This section includes:

- *Via file evaluation.*

A.1.1 Via file evaluation

When a tool that supports via files is invoked it:

1. Replaces the first specified `--via via_file` argument with the sequence of argument words extracted from the via file, including recursively processing any nested `--via` commands in the via file.
2. Processes any subsequent `--via via_file` arguments in the same way, in the order they are presented.

That is, via files are processed in the order you specify them, and each via file is processed completely including processing nested via files before processing the next via file.

A.2 Syntax

Via files must conform to the following syntax rules:

- A via file is a text file containing a sequence of words. Each word in the text file is converted into an argument string and passed to the tool.
- Words are separated by whitespace, or the end of a line, except in delimited strings. For example:
`--c90 --strict` (two words)
`--c90--strict` (one word)
- The end of a line is treated as whitespace. For example:
`--c90`
`--strict`
 is equivalent to:
`--c90 --strict`
- Strings enclosed in quotation marks ("), or apostrophes (') are treated as a single word. Within a quoted word, an apostrophe is treated as an ordinary character. Within an apostrophe delimited word, a quotation mark is treated as an ordinary character.
 Quotation marks are used to delimit filenames or path names that contain spaces. For example:
`-I C:\My Project\includes` (three words) `-I "C:\My Project\includes"` (two words)
 Apostrophes can be used to delimit words that contain quotes. For example:
`-DNAME='"RealView Compilation Tools"'` (one word)
- Characters enclosed in parentheses are treated as a single word. For example:
`--option(x, y, z)` (one word)
`--option (x, y, z)` (two words)
- Within quoted or apostrophe delimited strings, you can use a backslash (\) character to escape the quote, apostrophe, and backslash characters.
- A word that occurs immediately next to a delimited word is treated as a single word. For example:
`-I"C:\Project\includes"`
 is treated as the single word:
`-IC:\Project\includes`

- Lines beginning with a semicolon (;) or a hash (#) character as the first non whitespace character are comment lines. If a semicolon or hash character appears anywhere else in a line, it is not treated as the start of a comment. For example:

```
-o objectname.axf      ;this is not a comment
```

A comment ends at the end of a line, or at the end of the file. There are no multi-line comments, and there are no part-line comments.

- Lines that include the preprocessor option `-Dsymbol="value"` must be delimited with a single quote, either as `'-Dsymbol="value"'` or as `-Dsymbol='"value"'`. For example:

```
-c -DFOO_VALUE='"FOO_VALUE"'
```

Appendix B

Standard C Implementation Definition

This appendix gives information required by the ISO C standard for conforming C implementations. It contains the following section:

- *Implementation definition* on page B-2
- *Behaviors considered undefined by the ISO C Standard* on page B-9.

B.1 Implementation definition

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

Note

This appendix does not duplicate information that is part of Chapter 4 *Compiler-specific Features*. This appendix provides references where applicable.

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

Note

The support for the `wctype.h` and `wchar.h` headers excludes wide file operations.

B.1.1 Translation

Diagnostic messages produced by the compiler are of the form:

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

where *severity* is one of:

[blank] If the severity is blank, this is a remark and indicates common, but sometimes unconventional, use of C or C++. Remarks are not displayed by default. Use the `--remarks` option to display remark messages. See *Controlling the output of diagnostic messages* on page 5-4 for more information. Compilation continues.

Warning Flags unusual conditions in your code that might indicate a problem. Compilation continues.

Error Indicates a problem that causes the compilation to stop. For example, violations in the syntactic or semantic rules of the C or C++ language.

Internal fault

Indicates an internal problem with the compiler. Contact your supplier with the information listed in *Feedback* on page x.

Here:

error-code Is a number identifying the error type.

explanation Is a text description of the error.

See Chapter 5 *Diagnostic Messages* in the *Compiler User Guide* for more information.

B.1.2 Environment

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

- *main()*
- *Interactive device* on page B-4
- *Redirecting standard input, output, and error streams* on page B-4.

main()

The arguments given to `main()` are the words of the command line not including input/output redirections, delimited by whitespace, except where the whitespace 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 is not recognized inside double quotes.
-

Interactive device

In a non hosted 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.

Redirecting standard input, output, and error streams

Using the generic ARM C library, the standard input, output and error streams can be redirected at runtime. For example, if `mycopy` is a program running on a host debugger that 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:

<code>stdin</code>	The standard input stream is redirected to <code>infile</code> .
<code>stdout</code>	The standard output stream is redirected to <code>outfile</code> .
<code>stderr</code>	The standard error stream is redirected to <code>errfile</code> .

The permitted redirections are:

<code>0< filename</code>	Reads <code>stdin</code> from <code>filename</code> .
<code>< filename</code>	Reads <code>stdin</code> from <code>filename</code> .
<code>1> filename</code>	Writes <code>stdout</code> to <code>filename</code> .

`> filename` Writes stdout to *filename*.
`2> filename` Writes stderr to *filename*.
`2>&1` Writes stderr to the same place as stdout.
`>& file` Writes both stdout and stderr to *filename*.
`>> filename` Appends stdout to *filename*.
`>>& filename` Appends both stdout and stderr to *filename*.

To redirect stdin, stdout, and stderr on the target, you must define:

```
#pragma import(_main_redirection)
```

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()`.

B.1.3 Identifiers

See *Character sets and identifiers* on page 5-2 for more information.

B.1.4 Characters

See *Character sets and identifiers* on page 5-2 for more information.

B.1.5 Integers

See *Integer* on page 5-5 for more information.

B.1.6 Floating-point

See *Float* on page 5-5 for more information.

B.1.7 Arrays and pointers

See *Arrays and pointers* on page 5-5 for more information.

B.1.8 Registers

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

B.1.9 Structures, unions, enumerations, and bitfields

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.

See Chapter 5 *C and C++ Implementation Details* for more information.

Unions

See *Unions* on page 5-7 for information.

Enumerations

See *Enumerations* on page 5-7 for information.

Padding and alignment of structures

See *Structures* on page 5-8 for information.

Bitfields

See *Bitfields* on page 5-10 for information.

B.1.10 Qualifiers

An object that has a volatile-qualified type is accessed if any word or byte or halfword on ARM architectures that have halfword support, or it is read or written. For volatile-qualified objects, reads and writes occur as directly implied by the source code, in the order implied by the source code.

The effect of accessing a volatile-qualified **short** is undefined on ARM architectures that do not have halfword support.

B.1.11 Expression evaluation

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, t

he compiler can evaluate expressions in any order, regardless of parentheses. Therefore, 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 be varied by:

- the optimization level you are compiling at
- the release of the compiler you are using.

B.1.12 Preprocessing directives

The ISO standard C header files can be referred to as described in the standard, for example, `#include <stdio.h>`.

Quoted names for includable source files are supported. The compiler accepts 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 4-49.

B.1.13 Library functions

The ISO C library variants are listed in *About the runtime libraries* on page 1-2 in the *Libraries Guide*.

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 such as `printf`, an error might occur when the program is linked with the standard libraries. If it is not linked with standard libraries, no error is detected.
- The `__aeabi_assert()` function prints details of the failing diagnostic on `stderr` and then calls the `abort()` function:

```
*** assertion failed: expression, file name, line number
```

———— **Note** ————

The behavior of the `assert` macro depends on the conditions in operation at the most recent occurrence of `#include <assert.h>`. See *Exiting from the program* on page 2-37 in the *Libraries Guide* for more information.

For implementation details of mathematical functions, macros, locale, signals, and input/output see Chapter 2 *The C and C++ Libraries* in the *Libraries Guide*.

B.2 Behaviors considered undefined by the ISO C Standard

The following are considered undefined behavior by the ISO C Standard:

- In character and string escapes, if the character following the \ has no special meaning, the value of the escape is the character itself. For example, a warning is generated if you use \s because it is the same as s.
- A **struct** that has no named fields but at least one unnamed field is accepted by default, but generates an error in strict 1990 ISO Standard C.

Appendix C

Standard C++ Implementation Definition

The ARM compiler supports the majority of the language features described in the ISO/IEC standard for C++ when compiling C++. This appendix lists the C++ language features defined in the standard, and states whether or not that language feature is supported by ARM C++. It contains the following sections:

- *Integral conversion* on page C-2
- *Calling a pure virtual function* on page C-3
- *Major features of language support* on page C-4
- *Standard C++ library implementation definition* on page C-5.

Note

This appendix does not duplicate information that is part of the standard C implementation. See Appendix B *Standard C Implementation Definition*.

When compiling C++ in ISO C mode, the ARM compiler is 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 extensions to standard C++, see:

- *Standard C++ language extensions* on page 3-15
- *C99 language features available in C++ and C90* on page 3-7
- *Standard C and standard C++ language extensions* on page 3-19.

C.1 Integral conversion

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.

———— **Note** ————

This section is related to Section 4.7 of the ISO/IEC standard.

————

C.2 Calling a pure virtual function

Calling a pure virtual function is illegal. If your code calls a pure virtual function, then the compiler includes a call to the library function `__cxa_pure_virtual`.

`__cxa_pure_virtual` raises the signal **SIGPVFN**. The default signal handler prints an error message and exits. See `__default_signal_handler()` on page 2-79 in the *Libraries and Floating Point Support Guide* for more information.

C.3 Major features of language support

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

Table C-1 Major feature support for language

Major feature	ISO/IEC standard section	Support
Core language	1 to 13	Yes.
Templates	14	Yes, with the exception of export templates.
Exceptions	15	Yes.
Libraries	17 to 27	See the <i>Standard C++ library implementation definition</i> on page C-5 and the <i>Libraries and Floating Point Support Guide</i> .

C.4 Standard C++ library implementation definition

Version 2.02.03 of the Rogue Wave library provides a subset of the library defined in the standard. There are small differences from the 1999 ISO C standard. For information on the implementation definition, see *Standard C++ library implementation definition* on page 2-115 in the *Libraries and Floating Point Support Guide*.

The library can be used with user-defined functions to produce target-dependent applications. See *About the runtime libraries* on page 1-2 in the *Libraries and Floating Point Support Guide*.

Appendix D

C and C++ Compiler Implementation Limits

This appendix lists the implementation limits when using the ARM compiler to compile C and C++. It contains the following sections:

- *C++ ISO/IEC standard limits* on page D-2
- *Limits for integral numbers* on page D-4
- *Limits for floating-point numbers* on page D-5.

D.1 C++ ISO/IEC standard limits

The ISO/IEC C++ standard recommends minimum limits that a conforming compiler must accept. You must be aware of these when porting applications between compilers. Table D-1 gives a summary of these limits.

In this table, a limit of memory indicates that the ARM compiler imposes no limit, other than that imposed by the available memory.

Table D-1 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	memory
Number of initial characters in an external identifier.	1024	memory
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.	256	memory
Arguments in one function call.	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
Size of a C or C++ object (including arrays).	262 144	4294967 296
Nesting levels of #include file.	256	memory

Table D-1 Implementation limits (continued)

Description	Recommended	ARM
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 <code>atexit()</code> .	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

D.2 Limits for integral numbers

Table D-2 gives the ranges for integral numbers in ARM C and C++. The Endpoint column of the table gives the numerical value of the range endpoint. The Hex value column gives the bit pattern (in hexadecimal) that is interpreted as this value by the ARM compiler. These constants are defined in the `limits.h` include file.

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 information, as described in *Further reading* on page viii.

Table D-2 Integer ranges

Constant	Meaning	Value	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
LLONG_MAX	Maximum value of long long	9.2E+18	0x7FFFFFFF FFFFFFF
LLONG_MIN	Minimum value of long long	-9.2E+18	0x80000000 00000000
ULLONG_MAX	Maximum value of unsigned long long	1.8E+19	0xFFFFFFFF FFFFFFF

D.3 Limits for floating-point numbers

This section describes the characteristics of floating-point numbers.

Table D-3 gives the characteristics, ranges, and limits for floating-point numbers. These constants are defined in the `float.h` include file.

Table D-3 Floating-point limits

Constant	Meaning	Value
FLT_MAX	Maximum value of float	3.40282347e+38F
FLT_MIN	Minimum normalized positive floating-point number value of float	1.17549435e-38F
DBL_MAX	Maximum value of double	1.79769313486231571e+308
DBL_MIN	Minimum normalized positive floating-point number value of double	2.22507385850720138e-308
LDBL_MAX	Maximum value of long double	1.79769313486231571e+308
LDBL_MIN	Minimum normalized positive floating-point number value of long double	2.22507385850720138e-308
FLT_MAX_EXP	Maximum value of base 2 exponent for type float	128
FLT_MIN_EXP	Minimum value of base 2 exponent for type float	-125
DBL_MAX_EXP	Maximum value of base 2 exponent for type double	1024
DBL_MIN_EXP	Minimum value of base 2 exponent for type double	-1021
LDBL_MAX_EXP	Maximum value of base 2 exponent for type long double	1024
LDBL_MIN_EXP	Minimum value of base 2 exponent for type long double	-1021
FLT_MAX_10_EXP	Maximum value of base 10 exponent for type float	38
FLT_MIN_10_EXP	Minimum value of base 10 exponent for type float	-37
DBL_MAX_10_EXP	Maximum value of base 10 exponent for type double	308
DBL_MIN_10_EXP	Minimum value of base 10 exponent for type double	-307
LDBL_MAX_10_EXP	Maximum value of base 10 exponent for type long double	308
LDBL_MIN_10_EXP	Minimum value of base 10 exponent for type long double	-307

Table D-4 describes other characteristics of floating-point numbers. These constants are also defined in the `float.h` include file.

Table D-4 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 $1.0 + x \neq 1.0$ for type float	1.19209290e-7F
DBL_EPSILON	Smallest positive value of x that $1.0 + x \neq 1.0$ for type double	2.2204460492503131e-16
LDBL_EPSILON	Smallest positive value of x that $1.0 + x \neq 1.0$ for type long double	2.2204460492503131e-16L

———— **Note** —————

- When a floating-point number is converted to a shorter floating-point number, it is rounded to the nearest representable number.
- Floating-point arithmetic conforms to IEEE 754.

Appendix E

Using NEON Support

This appendix describes NEON intrinsics support in this release of the *RealView Compilation Tools* (RVCT).

This appendix contains the following sections:

- *Introduction* on page E-2
- *Vector data types* on page E-3
- *Intrinsics* on page E-4.

E.1 Introduction

RVCT provides intrinsics to generate NEON code for the Cortex-A8 processor in both ARM and Thumb state. The NEON intrinsics are defined in the header file `arm_neon.h`. The header file defines both the intrinsics and a set of vector types.

There is no support for NEON intrinsics for architectures before ARMv7. When building for earlier architectures, or for ARMv7 architecture profiles that do not include NEON, the compiler treats NEON intrinsics as ordinary function calls. This results in an error at link time.

E.2 Vector data types

The following types are defined to represent vectors. NEON vector data types are named according to the following pattern:

<type><size>x<number of lanes>_t

For example, `int16x4_t` is a vector containing four lanes each containing a signed 16-bit integer. Table E-1 lists the vector data types.

Table E-1 Vector data types

<code>int8x8_t</code>	<code>int8x16_t</code>
<code>int16x4_t</code>	<code>int16x8_t</code>
<code>int32x2_t</code>	<code>int32x4_t</code>
<code>int64x1_t</code>	<code>int64x2_t</code>
<code>uint8x8_t</code>	<code>uint8x16_t</code>
<code>uint16x4_t</code>	<code>uint16x8_t</code>
<code>uint32x2_t</code>	<code>uint32x4_t</code>
<code>uint64x1_t</code>	<code>uint64x2_t</code>
<code>float32x2_t</code>	<code>float32x4_t</code>
<code>poly8x8_t</code>	<code>poly8x16_t</code>
<code>poly16x4_t</code>	<code>poly16x8_t</code>

Some intrinsics use an array of vector types of the form:

<type><size>x<number of lanes>x<length of array>_t

These types are treated as ordinary C structures containing a single element named `val`.

An example structure definition is:

```
struct int16x4x2_t
{
    int16x4_t val[2];
};
```

There are array types defined for array lengths between 2 and 4, with any of the vector types listed above.

E.3 Intrinsics

The intrinsics described in this section map closely to NEON instructions. Each section begins with a list of function prototypes, with a comment specifying an equivalent assembler instruction. The compiler selects an instruction that has the required semantics, but there is no guarantee that the compiler produces the listed instruction.

The intrinsics use a naming scheme that is similar to the NEON unified assembler syntax. That is, each intrinsic has the form:

`<opname><flags>_<type>`

An additional `q` flag is provided to specify that the intrinsic operates on 128-bit vectors.

For example:

- `vmul_s16`, multiplies two vectors of signed 16-bit values.
This compiles to `VMUL.I16 d2, d0, d1`.
- `vaddl_u8`, is a long add of two 64-bit vectors containing unsigned 8-bit values, resulting in a 128-bit vector of unsigned 16-bit values.
This compiles to `VADDL.U8 q1, d0, d1`.

Note

The intrinsic function prototypes in this section use the following type annotations:

`__const(n)` the argument *n* must be a compile-time constant

`__constrange(min, max)`

the argument must be a compile-time constant in the range *min* to *max*

`__transfersize(n)`

the intrinsic loads *n* bytes from this pointer.

E.3.1 Addition

These intrinsics add vectors. Each lane in the result is the consequence of performing the addition on the corresponding lanes in each operand vector. The operations performed are as follows:

- *Vector add*: `vadd -> Vr[i]:=Va[i]+Vb[i]` on page E-5
- *Vector long add*: `vadd -> Vr[i]:=Va[i]+Vb[i]` on page E-5
- *Vector wide add*: `vadd -> Vr[i]:=Va[i]+Vb[i]` on page E-5
- *Vector halving add*: `vhadd -> Vr[i:]=(Va[i]+Vb[i])>>1` on page E-6
- *Vector rounding halving add*: `vrhadd -> Vr[i:]=(Va[i]+Vb[i]+1)>>1` on page E-6

- *Vector saturating add: vqadd -> Vr[i]:=sat<size>(Va[i]+Vb[i])* on page E-6
- *Vector add high half -> Vr[i]:=Va[i]+Vb[i]* on page E-7
- *Vector rounding add high half* on page E-7.

Vector add: vadd -> Vr[i]:=Va[i]+Vb[i]

Vr, Va, Vb have equal lane sizes.

```
int8x8_t   vadd_s8(int8x8_t a, int8x8_t b);      // VADD.I8 d0,d0,d0
int16x4_t  vadd_s16(int16x4_t a, int16x4_t b);   // VADD.I16 d0,d0,d0
int32x2_t  vadd_s32(int32x2_t a, int32x2_t b);   // VADD.I32 d0,d0,d0
int64x1_t  vadd_s64(int64x1_t a, int64x1_t b);   // VADD.I64 d0,d0,d0
float32x2_t vadd_f32(float32x2_t a, float32x2_t b); // VADD.F32 d0,d0,d0
uint8x8_t  vadd_u8(uint8x8_t a, uint8x8_t b);    // VADD.I8 d0,d0,d0
uint16x4_t vadd_u16(uint16x4_t a, uint16x4_t b); // VADD.I16 d0,d0,d0
uint32x2_t vadd_u32(uint32x2_t a, uint32x2_t b); // VADD.I32 d0,d0,d0
uint64x1_t vadd_u64(uint64x1_t a, uint64x1_t b); // VADD.I64 d0,d0,d0
int8x16_t  vaddq_s8(int8x16_t a, int8x16_t b);   // VADD.I8 q0,q0,q0
int16x8_t  vaddq_s16(int16x8_t a, int16x8_t b);  // VADD.I16 q0,q0,q0
int32x4_t  vaddq_s32(int32x4_t a, int32x4_t b);  // VADD.I32 q0,q0,q0
int64x2_t  vaddq_s64(int64x2_t a, int64x2_t b);  // VADD.I64 q0,q0,q0
float32x4_t vaddq_f32(float32x4_t a, float32x4_t b); // VADD.F32 q0,q0,q0
uint8x16_t vaddq_u8(uint8x16_t a, uint8x16_t b); // VADD.I8 q0,q0,q0
uint16x8_t vaddq_u16(uint16x8_t a, uint16x8_t b); // VADD.I16 q0,q0,q0
uint32x4_t vaddq_u32(uint32x4_t a, uint32x4_t b); // VADD.I32 q0,q0,q0
uint64x2_t vaddq_u64(uint64x2_t a, uint64x2_t b); // VADD.I64 q0,q0,q0
```

Vector long add: vaddl -> Vr[i]:=Va[i]+Vb[i]

Va, Vb have equal lane sizes, result is a 128 bit vector of lanes that are twice the width.

```
int16x8_t  vaddl_s8(int8x8_t a, int8x8_t b);     // VADDL.S8 q0,d0,d0
int32x4_t  vaddl_s16(int16x4_t a, int16x4_t b);  // VADDL.S16 q0,d0,d0
int64x2_t  vaddl_s32(int32x2_t a, int32x2_t b);  // VADDL.S32 q0,d0,d0
uint16x8_t vaddl_u8(uint8x8_t a, uint8x8_t b);   // VADDL.U8 q0,d0,d0
uint32x4_t vaddl_u16(uint16x4_t a, uint16x4_t b); // VADDL.U16 q0,d0,d0
uint64x2_t vaddl_u32(uint32x2_t a, uint32x2_t b); // VADDL.U32 q0,d0,d0
```

Vector wide add: vaddw -> Vr[i]:=Va[i]+Vb[i]

```
int16x8_t  vaddw_s8(int16x8_t a, int8x8_t b);    // VADDW.S8 q0,q0,d0
int32x4_t  vaddw_s16(int32x4_t a, int16x4_t b);  // VADDW.S16 q0,q0,d0
int64x2_t  vaddw_s32(int64x2_t a, int32x2_t b);  // VADDW.S32 q0,q0,d0
uint16x8_t vaddw_u8(uint16x8_t a, uint8x8_t b);  // VADDW.U8 q0,q0,d0
uint32x4_t vaddw_u16(uint32x4_t a, uint16x4_t b); // VADDW.U16 q0,q0,d0
uint64x2_t vaddw_u32(uint64x2_t a, uint32x2_t b); // VADDW.U32 q0,q0,d0
```

Vector halving add: vhadd -> Vr[i]:=(Va[i]+Vb[i])>>1

```

int8x8_t   vhadd_s8(int8x8_t a, int8x8_t b);           // VHADD.S8 d0,d0,d0
int16x4_t  vhadd_s16(int16x4_t a, int16x4_t b);        // VHADD.S16 d0,d0,d0
int32x2_t  vhadd_s32(int32x2_t a, int32x2_t b);        // VHADD.S32 d0,d0,d0
uint8x8_t  vhadd_u8(uint8x8_t a, uint8x8_t b);         // VHADD.U8 d0,d0,d0
uint16x4_t vhadd_u16(uint16x4_t a, uint16x4_t b);      // VHADD.U16 d0,d0,d0
uint32x2_t vhadd_u32(uint32x2_t a, uint32x2_t b);      // VHADD.U32 d0,d0,d0
int8x16_t  vhaddq_s8(int8x16_t a, int8x16_t b);        // VHADD.S8 q0,q0,q0
int16x8_t  vhaddq_s16(int16x8_t a, int16x8_t b);       // VHADD.S16 q0,q0,q0
int32x4_t  vhaddq_s32(int32x4_t a, int32x4_t b);       // VHADD.S32 q0,q0,q0
uint8x16_t vhaddq_u8(uint8x16_t a, uint8x16_t b);      // VHADD.U8 q0,q0,q0
uint16x8_t vhaddq_u16(uint16x8_t a, uint16x8_t b);     // VHADD.U16 q0,q0,q0
uint32x4_t vhaddq_u32(uint32x4_t a, uint32x4_t b);     // VHADD.U32 q0,q0,q0

```

Vector rounding halving add: vrhadd -> Vr[i]:=(Va[i]+Vb[i]+1)>>1

```

int8x8_t   vrhadd_s8(int8x8_t a, int8x8_t b);          // VRHADD.S8 d0,d0,d0
int16x4_t  vrhadd_s16(int16x4_t a, int16x4_t b);       // VRHADD.S16 d0,d0,d0
int32x2_t  vrhadd_s32(int32x2_t a, int32x2_t b);       // VRHADD.S32 d0,d0,d0
uint8x8_t  vrhadd_u8(uint8x8_t a, uint8x8_t b);        // VRHADD.U8 d0,d0,d0
uint16x4_t vrhadd_u16(uint16x4_t a, uint16x4_t b);     // VRHADD.U16 d0,d0,d0
uint32x2_t vrhadd_u32(uint32x2_t a, uint32x2_t b);     // VRHADD.U32 d0,d0,d0
int8x16_t  vrhaddq_s8(int8x16_t a, int8x16_t b);       // VRHADD.S8 q0,q0,q0
int16x8_t  vrhaddq_s16(int16x8_t a, int16x8_t b);      // VRHADD.S16 q0,q0,q0
int32x4_t  vrhaddq_s32(int32x4_t a, int32x4_t b);      // VRHADD.S32 q0,q0,q0
uint8x16_t vrhaddq_u8(uint8x16_t a, uint8x16_t b);     // VRHADD.U8 q0,q0,q0
uint16x8_t vrhaddq_u16(uint16x8_t a, uint16x8_t b);    // VRHADD.U16 q0,q0,q0
uint32x4_t vrhaddq_u32(uint32x4_t a, uint32x4_t b);    // VRHADD.U32 q0,q0,q0

```

Vector saturating add: vqadd -> Vr[i]:=sat<size>(Va[i]+Vb[i])

```

int8x8_t   vqadd_s8(int8x8_t a, int8x8_t b);           // VQADD.S8 d0,d0,d0
int16x4_t  vqadd_s16(int16x4_t a, int16x4_t b);        // VQADD.S16 d0,d0,d0
int32x2_t  vqadd_s32(int32x2_t a, int32x2_t b);        // VQADD.S32 d0,d0,d0
int64x1_t  vqadd_s64(int64x1_t a, int64x1_t b);        // VQADD.S64 d0,d0,d0
uint8x8_t  vqadd_u8(uint8x8_t a, uint8x8_t b);         // VQADD.U8 d0,d0,d0
uint16x4_t vqadd_u16(uint16x4_t a, uint16x4_t b);      // VQADD.U16 d0,d0,d0
uint32x2_t vqadd_u32(uint32x2_t a, uint32x2_t b);      // VQADD.U32 d0,d0,d0
uint64x1_t vqadd_u64(uint64x1_t a, uint64x1_t b);      // VQADD.U64 d0,d0,d0
int8x16_t  vqaddq_s8(int8x16_t a, int8x16_t b);        // VQADD.S8 q0,q0,q0
int16x8_t  vqaddq_s16(int16x8_t a, int16x8_t b);       // VQADD.S16 q0,q0,q0
int32x4_t  vqaddq_s32(int32x4_t a, int32x4_t b);       // VQADD.S32 q0,q0,q0
int64x2_t  vqaddq_s64(int64x2_t a, int64x2_t b);       // VQADD.S64 q0,q0,q0
uint8x16_t vqaddq_u8(uint8x16_t a, uint8x16_t b);      // VQADD.U8 q0,q0,q0
uint16x8_t vqaddq_u16(uint16x8_t a, uint16x8_t b);     // VQADD.U16 q0,q0,q0
uint32x4_t vqaddq_u32(uint32x4_t a, uint32x4_t b);     // VQADD.U32 q0,q0,q0
uint64x2_t vqaddq_u64(uint64x2_t a, uint64x2_t b);     // VQADD.U64 q0,q0,q0

```

Vector add high half -> Vr[i]:=Va[i]+Vb[i]

```

int8x8_t   vaddhn_s16(int16x8_t a, int16x8_t b);    // VADDHN.I16 d0,q0,q0
int16x4_t  vaddhn_s32(int32x4_t a, int32x4_t b);    // VADDHN.I32 d0,q0,q0
int32x2_t  vaddhn_s64(int64x2_t a, int64x2_t b);    // VADDHN.I64 d0,q0,q0
uint8x8_t  vaddhn_u16(uint16x8_t a, uint16x8_t b);  // VADDHN.I16 d0,q0,q0
uint16x4_t vaddhn_u32(uint32x4_t a, uint32x4_t b);  // VADDHN.I32 d0,q0,q0
uint32x2_t vaddhn_u64(uint64x2_t a, uint64x2_t b);  // VADDHN.I64 d0,q0,q0

```

Vector rounding add high half

```

int8x8_t   vraddhn_s16(int16x8_t a, int16x8_t b);   // VRADDHN.I16 d0,q0,q0
int16x4_t  vraddhn_s32(int32x4_t a, int32x4_t b);   // VRADDHN.I32 d0,q0,q0
int32x2_t  vraddhn_s64(int64x2_t a, int64x2_t b);   // VRADDHN.I64 d0,q0,q0
uint8x8_t  vraddhn_u16(uint16x8_t a, uint16x8_t b); // VRADDHN.I16 d0,q0,q0
uint16x4_t vraddhn_u32(uint32x4_t a, uint32x4_t b); // VRADDHN.I32 d0,q0,q0
uint32x2_t vraddhn_u64(uint64x2_t a, uint64x2_t b); // VRADDHN.I64 d0,q0,q0

```

E.3.2 Multiplication

These intrinsics provide operations including multiplication.

Vector multiply: vmul -> Vr[i] := Va[i] * Vb[i]

```

int8x8_t   vmul_s8(int8x8_t a, int8x8_t b);          // VMUL.I8 d0,d0,d0
int16x4_t  vmul_s16(int16x4_t a, int16x4_t b);       // VMUL.I16 d0,d0,d0
int32x2_t  vmul_s32(int32x2_t a, int32x2_t b);       // VMUL.I32 d0,d0,d0
float32x2_t vmul_f32(float32x2_t a, float32x2_t b);  // VMUL.F32 d0,d0,d0
uint8x8_t  vmul_u8(uint8x8_t a, uint8x8_t b);       // VMUL.I8 d0,d0,d0
uint16x4_t vmul_u16(uint16x4_t a, uint16x4_t b);    // VMUL.I16 d0,d0,d0
uint32x2_t vmul_u32(uint32x2_t a, uint32x2_t b);    // VMUL.I32 d0,d0,d0
poly8x8_t  vmul_p8(poly8x8_t a, poly8x8_t b);       // VMUL.P8 d0,d0,d0
int8x16_t  vmulq_s8(int8x16_t a, int8x16_t b);      // VMUL.I8 q0,q0,q0
int16x8_t  vmulq_s16(int16x8_t a, int16x8_t b);     // VMUL.I16 q0,q0,q0
int32x4_t  vmulq_s32(int32x4_t a, int32x4_t b);     // VMUL.I32 q0,q0,q0
float32x4_t vmulq_f32(float32x4_t a, float32x4_t b); // VMUL.F32 q0,q0,q0
uint8x16_t vmulq_u8(uint8x16_t a, uint8x16_t b);    // VMUL.I8 q0,q0,q0
uint16x8_t vmulq_u16(uint16x8_t a, uint16x8_t b);   // VMUL.I16 q0,q0,q0
uint32x4_t vmulq_u32(uint32x4_t a, uint32x4_t b);   // VMUL.I32 q0,q0,q0
poly8x16_t vmulq_p8(poly8x16_t a, poly8x16_t b);    // VMUL.P8 q0,q0,q0

```

Vector multiply accumulate: vmla -> Vr[i] := Va[i] + Vb[i] * Vc[i]

```

int8x8_t   vmla_s8(int8x8_t a, int8x8_t b, int8x8_t c); // VMLA.I8 d0,d0,d0
int16x4_t  vmla_s16(int16x4_t a, int16x4_t b, int16x4_t c); // VMLA.I16 d0,d0,d0
int32x2_t  vmla_s32(int32x2_t a, int32x2_t b, int32x2_t c); // VMLA.I32 d0,d0,d0
float32x2_t vmla_f32(float32x2_t a, float32x2_t b, float32x2_t c); // VMLA.F32 d0,d0,d0
uint8x8_t  vmla_u8(uint8x8_t a, uint8x8_t b, uint8x8_t c); // VMLA.I8 d0,d0,d0

```

```

uint16x4_t  vmla_u16(uint16x4_t a, uint16x4_t b, uint16x4_t c);      // VMLA.I16 d0,d0,d0
uint32x2_t  vmla_u32(uint32x2_t a, uint32x2_t b, uint32x2_t c);      // VMLA.I32 d0,d0,d0
int8x16_t   vmlaq_s8(int8x16_t a, int8x16_t b, int8x16_t c);        // VMLA.I8 q0,q0,q0
int16x8_t   vmlaq_s16(int16x8_t a, int16x8_t b, int16x8_t c);        // VMLA.I16 q0,q0,q0
int32x4_t   vmlaq_s32(int32x4_t a, int32x4_t b, int32x4_t c);        // VMLA.I32 q0,q0,q0
float32x4_t vmlaq_f32(float32x4_t a, float32x4_t b, float32x4_t c);  // VMLA.F32 q0,q0,q0
uint8x16_t  vmlaq_u8(uint8x16_t a, uint8x16_t b, uint8x16_t c);      // VMLA.I8 q0,q0,q0
uint16x8_t  vmlaq_u16(uint16x8_t a, uint16x8_t b, uint16x8_t c);     // VMLA.I16 q0,q0,q0
uint32x4_t  vmlaq_u32(uint32x4_t a, uint32x4_t b, uint32x4_t c);     // VMLA.I32 q0,q0,q0

```

Vector multiply accumulate long: vmla -> $Vr[i] := Va[i] + Vb[i] * Vc[i]$

```

int16x8_t  vmlal_s8(int16x8_t a, int8x8_t b, int8x8_t c);           // VMLAL.S8 q0,d0,d0
int32x4_t  vmlal_s16(int32x4_t a, int16x4_t b, int16x4_t c);        // VMLAL.S16 q0,d0,d0
int64x2_t  vmlal_s32(int64x2_t a, int32x2_t b, int32x2_t c);        // VMLAL.S32 q0,d0,d0
uint16x8_t vmlal_u8(uint16x8_t a, uint8x8_t b, uint8x8_t c);        // VMLAL.U8 q0,d0,d0
uint32x4_t vmlal_u16(uint32x4_t a, uint16x4_t b, uint16x4_t c);     // VMLAL.U16 q0,d0,d0
uint64x2_t vmlal_u32(uint64x2_t a, uint32x2_t b, uint32x2_t c);     // VMLAL.U32 q0,d0,d0

```

Vector multiply subtract: vmls -> $Vr[i] := Va[i] - Vb[i] * Vc[i]$

```

int8x8_t   vmls_s8(int8x8_t a, int8x8_t b, int8x8_t c);           // VMLS.I8 d0,d0,d0
int16x4_t  vmls_s16(int16x4_t a, int16x4_t b, int16x4_t c);        // VMLS.I16 d0,d0,d0
int32x2_t  vmls_s32(int32x2_t a, int32x2_t b, int32x2_t c);        // VMLS.I32 d0,d0,d0
float32x2_t vmls_f32(float32x2_t a, float32x2_t b, float32x2_t c);  // VMLS.F32 d0,d0,d0
uint8x8_t  vmls_u8(uint8x8_t a, uint8x8_t b, uint8x8_t c);         // VMLS.I8 d0,d0,d0
uint16x4_t vmls_u16(uint16x4_t a, uint16x4_t b, uint16x4_t c);     // VMLS.I16 d0,d0,d0
uint32x2_t vmls_u32(uint32x2_t a, uint32x2_t b, uint32x2_t c);     // VMLS.I32 d0,d0,d0
int8x16_t  vmlsq_s8(int8x16_t a, int8x16_t b, int8x16_t c);        // VMLS.I8 q0,q0,q0
int16x8_t  vmlsq_s16(int16x8_t a, int16x8_t b, int16x8_t c);        // VMLS.I16 q0,q0,q0
int32x4_t  vmlsq_s32(int32x4_t a, int32x4_t b, int32x4_t c);        // VMLS.I32 q0,q0,q0
float32x4_t vmlsq_f32(float32x4_t a, float32x4_t b, float32x4_t c);  // VMLS.F32 q0,q0,q0
uint8x16_t vmlsq_u8(uint8x16_t a, uint8x16_t b, uint8x16_t c);      // VMLS.I8 q0,q0,q0
uint16x8_t vmlsq_u16(uint16x8_t a, uint16x8_t b, uint16x8_t c);     // VMLS.I16 q0,q0,q0
uint32x4_t vmlsq_u32(uint32x4_t a, uint32x4_t b, uint32x4_t c);     // VMLS.I32 q0,q0,q0

```

Vector multiply subtract long

```

int16x8_t  vmlsl_s8(int16x8_t a, int8x8_t b, int8x8_t c);           // VMLSL.S8 q0,d0,d0
int32x4_t  vmlsl_s16(int32x4_t a, int16x4_t b, int16x4_t c);        // VMLSL.S16 q0,d0,d0
int64x2_t  vmlsl_s32(int64x2_t a, int32x2_t b, int32x2_t c);        // VMLSL.S32 q0,d0,d0
uint16x8_t vmlsl_u8(uint16x8_t a, uint8x8_t b, uint8x8_t c);        // VMLSL.U8 q0,d0,d0
uint32x4_t vmlsl_u16(uint32x4_t a, uint16x4_t b, uint16x4_t c);     // VMLSL.U16 q0,d0,d0
uint64x2_t vmlsl_u32(uint64x2_t a, uint32x2_t b, uint32x2_t c);     // VMLSL.U32 q0,d0,d0

```

Vector saturating doubling multiply high

```
int16x4_t vqdmulh_s16(int16x4_t a, int16x4_t b); // VQDMULH.S16 d0,d0,d0
int32x2_t vqdmulh_s32(int32x2_t a, int32x2_t b); // VQDMULH.S32 d0,d0,d0
int16x8_t vqdmulhq_s16(int16x8_t a, int16x8_t b); // VQDMULH.S16 q0,q0,q0
int32x4_t vqdmulhq_s32(int32x4_t a, int32x4_t b); // VQDMULH.S32 q0,q0,q0
```

Vector saturating rounding doubling multiply high

```
int16x4_t vqrdmulh_s16(int16x4_t a, int16x4_t b); // VQRDMULH.S16 d0,d0,d0
int32x2_t vqrdmulh_s32(int32x2_t a, int32x2_t b); // VQRDMULH.S32 d0,d0,d0
int16x8_t vqrdmulhq_s16(int16x8_t a, int16x8_t b); // VQRDMULH.S16 q0,q0,q0
int32x4_t vqrdmulhq_s32(int32x4_t a, int32x4_t b); // VQRDMULH.S32 q0,q0,q0
```

Vector saturating doubling multiply accumulate long

```
int32x4_t vqdm1al_s16(int32x4_t a, int16x4_t b, int16x4_t c); // VQDMLAL.S16 q0,d0,d0
int64x2_t vqdm1al_s32(int64x2_t a, int32x2_t b, int32x2_t c); // VQDMLAL.S32 q0,d0,d0
```

Vector saturating doubling multiply subtract long

```
int32x4_t vqdm1sl_s16(int32x4_t a, int16x4_t b, int16x4_t c); // VQDMLSL.S16 q0,d0,d0
int64x2_t vqdm1sl_s32(int64x2_t a, int32x2_t b, int32x2_t c); // VQDMLSL.S32 q0,d0,d0
```

Vector long multiply

```
int16x8_t vmull_s8(int8x8_t a, int8x8_t b); // VMULL.S8 q0,d0,d0
int32x4_t vmull_s16(int16x4_t a, int16x4_t b); // VMULL.S16 q0,d0,d0
int64x2_t vmull_s32(int32x2_t a, int32x2_t b); // VMULL.S32 q0,d0,d0
uint16x8_t vmull_u8(uint8x8_t a, uint8x8_t b); // VMULL.U8 q0,d0,d0
uint32x4_t vmull_u16(uint16x4_t a, uint16x4_t b); // VMULL.U16 q0,d0,d0
uint64x2_t vmull_u32(uint32x2_t a, uint32x2_t b); // VMULL.U32 q0,d0,d0
poly16x8_t vmull_p8(poly8x8_t a, poly8x8_t b); // VMULL.P8 q0,d0,d0
```

Vector saturating doubling long multiply

```
int32x4_t vqdmull_s16(int16x4_t a, int16x4_t b); // VQDMULL.S16 q0,d0,d0
int64x2_t vqdmull_s32(int32x2_t a, int32x2_t b); // VQDMULL.S32 q0,d0,d0
```

E.3.3 Subtraction

These intrinsics provide operations including subtraction.

Vector subtract

```

int8x8_t    vsub_s8(int8x8_t a, int8x8_t b);           // VSUB.I8 d0,d0,d0
int16x4_t   vsub_s16(int16x4_t a, int16x4_t b);        // VSUB.I16 d0,d0,d0
int32x2_t   vsub_s32(int32x2_t a, int32x2_t b);        // VSUB.I32 d0,d0,d0
int64x1_t   vsub_s64(int64x1_t a, int64x1_t b);        // VSUB.I64 d0,d0,d0
float32x2_t vsub_f32(float32x2_t a, float32x2_t b);    // VSUB.F32 d0,d0,d0
uint8x8_t   vsub_u8(uint8x8_t a, uint8x8_t b);         // VSUB.I8 d0,d0,d0
uint16x4_t  vsub_u16(uint16x4_t a, uint16x4_t b);      // VSUB.I16 d0,d0,d0
uint32x2_t  vsub_u32(uint32x2_t a, uint32x2_t b);      // VSUB.I32 d0,d0,d0
uint64x1_t  vsub_u64(uint64x1_t a, uint64x1_t b);      // VSUB.I64 d0,d0,d0
int8x16_t   vsubq_s8(int8x16_t a, int8x16_t b);        // VSUB.I8 q0,q0,q0
int16x8_t   vsubq_s16(int16x8_t a, int16x8_t b);       // VSUB.I16 q0,q0,q0
int32x4_t   vsubq_s32(int32x4_t a, int32x4_t b);       // VSUB.I32 q0,q0,q0
int64x2_t   vsubq_s64(int64x2_t a, int64x2_t b);       // VSUB.I64 q0,q0,q0
float32x4_t vsubq_f32(float32x4_t a, float32x4_t b);    // VSUB.F32 q0,q0,q0
uint8x16_t  vsubq_u8(uint8x16_t a, uint8x16_t b);      // VSUB.I8 q0,q0,q0
uint16x8_t  vsubq_u16(uint16x8_t a, uint16x8_t b);     // VSUB.I16 q0,q0,q0
uint32x4_t  vsubq_u32(uint32x4_t a, uint32x4_t b);     // VSUB.I32 q0,q0,q0
uint64x2_t  vsubq_u64(uint64x2_t a, uint64x2_t b);     // VSUB.I64 q0,q0,q0

```

Vector long subtract: vsubl -> Vr[i]:=Va[i]+Vb[i]

```

int16x8_t   vsubl_s8(int8x8_t a, int8x8_t b);          // VSUBL.S8 q0,d0,d0
int32x4_t   vsubl_s16(int16x4_t a, int16x4_t b);       // VSUBL.S16 q0,d0,d0
int64x2_t   vsubl_s32(int32x2_t a, int32x2_t b);       // VSUBL.S32 q0,d0,d0
uint16x8_t  vsubl_u8(uint8x8_t a, uint8x8_t b);        // VSUBL.U8 q0,d0,d0
uint32x4_t  vsubl_u16(uint16x4_t a, uint16x4_t b);     // VSUBL.U16 q0,d0,d0
uint64x2_t  vsubl_u32(uint32x2_t a, uint32x2_t b);     // VSUBL.U32 q0,d0,d0

```

Vector wide subtract: vsubw -> Vr[i]:=Va[i]+Vb[i]

```

int16x8_t   vsubw_s8(int16x8_t a, int8x8_t b);         // VSUBW.S8 q0,q0,d0
int32x4_t   vsubw_s16(int32x4_t a, int16x4_t b);       // VSUBW.S16 q0,q0,d0
int64x2_t   vsubw_s32(int64x2_t a, int32x2_t b);       // VSUBW.S32 q0,q0,d0
uint16x8_t  vsubw_u8(uint16x8_t a, uint8x8_t b);       // VSUBW.U8 q0,q0,d0
uint32x4_t  vsubw_u16(uint32x4_t a, uint16x4_t b);     // VSUBW.U16 q0,q0,d0
uint64x2_t  vsubw_u32(uint64x2_t a, uint32x2_t b);     // VSUBW.U32 q0,q0,d0

```

Vector saturating subtract

```

int8x8_t    vqsub_s8(int8x8_t a, int8x8_t b);          // VQSUB.S8 d0,d0,d0
int16x4_t   vqsub_s16(int16x4_t a, int16x4_t b);       // VQSUB.S16 d0,d0,d0
int32x2_t   vqsub_s32(int32x2_t a, int32x2_t b);       // VQSUB.S32 d0,d0,d0
int64x1_t   vqsub_s64(int64x1_t a, int64x1_t b);       // VQSUB.S64 d0,d0,d0
uint8x8_t   vqsub_u8(uint8x8_t a, uint8x8_t b);        // VQSUB.U8 d0,d0,d0
uint16x4_t  vqsub_u16(uint16x4_t a, uint16x4_t b);     // VQSUB.U16 d0,d0,d0
uint32x2_t  vqsub_u32(uint32x2_t a, uint32x2_t b);     // VQSUB.U32 d0,d0,d0

```

```

uint64x1_t vqsub_u64(uint64x1_t a, uint64x1_t b); // VQSUB.U64 d0,d0,d0
int8x16_t vqsubq_s8(int8x16_t a, int8x16_t b); // VQSUB.S8 q0,q0,q0
int16x8_t vqsubq_s16(int16x8_t a, int16x8_t b); // VQSUB.S16 q0,q0,q0
int32x4_t vqsubq_s32(int32x4_t a, int32x4_t b); // VQSUB.S32 q0,q0,q0
int64x2_t vqsubq_s64(int64x2_t a, int64x2_t b); // VQSUB.S64 q0,q0,q0
uint8x16_t vqsubq_u8(uint8x16_t a, uint8x16_t b); // VQSUB.U8 q0,q0,q0
uint16x8_t vqsubq_u16(uint16x8_t a, uint16x8_t b); // VQSUB.U16 q0,q0,q0
uint32x4_t vqsubq_u32(uint32x4_t a, uint32x4_t b); // VQSUB.U32 q0,q0,q0
uint64x2_t vqsubq_u64(uint64x2_t a, uint64x2_t b); // VQSUB.U64 q0,q0,q0

```

Vector halving subtract

```

int8x8_t vhsb_s8(int8x8_t a, int8x8_t b); // VHSUB.S8 d0,d0,d0
int16x4_t vhsb_s16(int16x4_t a, int16x4_t b); // VHSUB.S16 d0,d0,d0
int32x2_t vhsb_s32(int32x2_t a, int32x2_t b); // VHSUB.S32 d0,d0,d0
uint8x8_t vhsb_u8(uint8x8_t a, uint8x8_t b); // VHSUB.U8 d0,d0,d0
uint16x4_t vhsb_u16(uint16x4_t a, uint16x4_t b); // VHSUB.U16 d0,d0,d0
uint32x2_t vhsb_u32(uint32x2_t a, uint32x2_t b); // VHSUB.U32 d0,d0,d0
int8x16_t vhsbq_s8(int8x16_t a, int8x16_t b); // VHSUB.S8 q0,q0,q0
int16x8_t vhsbq_s16(int16x8_t a, int16x8_t b); // VHSUB.S16 q0,q0,q0
int32x4_t vhsbq_s32(int32x4_t a, int32x4_t b); // VHSUB.S32 q0,q0,q0
uint8x16_t vhsbq_u8(uint8x16_t a, uint8x16_t b); // VHSUB.U8 q0,q0,q0
uint16x8_t vhsbq_u16(uint16x8_t a, uint16x8_t b); // VHSUB.U16 q0,q0,q0
uint32x4_t vhsbq_u32(uint32x4_t a, uint32x4_t b); // VHSUB.U32 q0,q0,q0

```

Vector subtract high half

```

int8x8_t vsubhn_s16(int16x8_t a, int16x8_t b); // VSUBHN.I16 d0,q0,q0
int16x4_t vsubhn_s32(int32x4_t a, int32x4_t b); // VSUBHN.I32 d0,q0,q0
int32x2_t vsubhn_s64(int64x2_t a, int64x2_t b); // VSUBHN.I64 d0,q0,q0
uint8x8_t vsubhn_u16(uint16x8_t a, uint16x8_t b); // VSUBHN.I16 d0,q0,q0
uint16x4_t vsubhn_u32(uint32x4_t a, uint32x4_t b); // VSUBHN.I32 d0,q0,q0
uint32x2_t vsubhn_u64(uint64x2_t a, uint64x2_t b); // VSUBHN.I64 d0,q0,q0

```

Vector rounding subtract high half

```

int8x8_t vrsbhn_s16(int16x8_t a, int16x8_t b); // VRSUBHN.I16 d0,q0,q0
int16x4_t vrsbhn_s32(int32x4_t a, int32x4_t b); // VRSUBHN.I32 d0,q0,q0
int32x2_t vrsbhn_s64(int64x2_t a, int64x2_t b); // VRSUBHN.I64 d0,q0,q0
uint8x8_t vrsbhn_u16(uint16x8_t a, uint16x8_t b); // VRSUBHN.I16 d0,q0,q0
uint16x4_t vrsbhn_u32(uint32x4_t a, uint32x4_t b); // VRSUBHN.I32 d0,q0,q0
uint32x2_t vrsbhn_u64(uint64x2_t a, uint64x2_t b); // VRSUBHN.I64 d0,q0,q0

```

E.3.4 Comparison

A range of comparison intrinsics are provided. If the comparison is true for a lane, the result in that lane is all bits set to one. If the comparison is false for a lane, all bits are set to zero. The return type is an unsigned integer type. This means that you can use the result of a comparison as the first argument for the vbsl intrinsics.

Vector compare equal

```
uint8x8_t vceq_s8(int8x8_t a, int8x8_t b);           // VCEQ.I8 d0, d0, d0
uint16x4_t vceq_s16(int16x4_t a, int16x4_t b);       // VCEQ.I16 d0, d0, d0
uint32x2_t vceq_s32(int32x2_t a, int32x2_t b);       // VCEQ.I32 d0, d0, d0
uint32x2_t vceq_f32(float32x2_t a, float32x2_t b);   // VCEQ.F32 d0, d0, d0
uint8x8_t vceq_u8(uint8x8_t a, uint8x8_t b);        // VCEQ.I8 d0, d0, d0
uint16x4_t vceq_u16(uint16x4_t a, uint16x4_t b);     // VCEQ.I16 d0, d0, d0
uint32x2_t vceq_u32(uint32x2_t a, uint32x2_t b);     // VCEQ.I32 d0, d0, d0
uint8x8_t vceq_p8(poly8x8_t a, poly8x8_t b);        // VCEQ.I8 d0, d0, d0
uint8x16_t vceqq_s8(int8x16_t a, int8x16_t b);      // VCEQ.I8 q0, q0, q0
uint16x8_t vceqq_s16(int16x8_t a, int16x8_t b);     // VCEQ.I16 q0, q0, q0
uint32x4_t vceqq_s32(int32x4_t a, int32x4_t b);     // VCEQ.I32 q0, q0, q0
uint32x4_t vceqq_f32(float32x4_t a, float32x4_t b); // VCEQ.F32 q0, q0, q0
uint8x16_t vceqq_u8(uint8x16_t a, uint8x16_t b);    // VCEQ.I8 q0, q0, q0
uint16x8_t vceqq_u16(uint16x8_t a, uint16x8_t b);   // VCEQ.I16 q0, q0, q0
uint32x4_t vceqq_u32(uint32x4_t a, uint32x4_t b);   // VCEQ.I32 q0, q0, q0
uint8x16_t vceqq_p8(poly8x16_t a, poly8x16_t b);    // VCEQ.I8 q0, q0, q0
```

Vector compare greater-than or equal

```
uint8x8_t vcge_s8(int8x8_t a, int8x8_t b);           // VCGE.S8 d0, d0, d0
uint16x4_t vcge_s16(int16x4_t a, int16x4_t b);       // VCGE.S16 d0, d0, d0
uint32x2_t vcge_s32(int32x2_t a, int32x2_t b);       // VCGE.S32 d0, d0, d0
uint32x2_t vcge_f32(float32x2_t a, float32x2_t b);   // VCGE.F32 d0, d0, d0
uint8x8_t vcge_u8(uint8x8_t a, uint8x8_t b);        // VCGE.U8 d0, d0, d0
uint16x4_t vcge_u16(uint16x4_t a, uint16x4_t b);     // VCGE.U16 d0, d0, d0
uint32x2_t vcge_u32(uint32x2_t a, uint32x2_t b);     // VCGE.U32 d0, d0, d0
uint8x16_t vcgeq_s8(int8x16_t a, int8x16_t b);      // VCGE.S8 q0, q0, q0
uint16x8_t vcgeq_s16(int16x8_t a, int16x8_t b);     // VCGE.S16 q0, q0, q0
uint32x4_t vcgeq_s32(int32x4_t a, int32x4_t b);     // VCGE.S32 q0, q0, q0
uint32x4_t vcgeq_f32(float32x4_t a, float32x4_t b); // VCGE.F32 q0, q0, q0
uint8x16_t vcgeq_u8(uint8x16_t a, uint8x16_t b);    // VCGE.U8 q0, q0, q0
uint16x8_t vcgeq_u16(uint16x8_t a, uint16x8_t b);   // VCGE.U16 q0, q0, q0
uint32x4_t vcgeq_u32(uint32x4_t a, uint32x4_t b);   // VCGE.U32 q0, q0, q0
```

Vector compare less-than or equal

```
uint8x8_t vcle_s8(int8x8_t a, int8x8_t b);           // VCGE.S8 d0, d0, d0
uint16x4_t vcle_s16(int16x4_t a, int16x4_t b);       // VCGE.S16 d0, d0, d0
uint32x2_t vcle_s32(int32x2_t a, int32x2_t b);       // VCGE.S32 d0, d0, d0
```

```

uint32x2_t vcle_f32(float32x2_t a, float32x2_t b); // VCGE.F32 d0, d0, d0
uint8x8_t  vcle_u8(uint8x8_t a, uint8x8_t b);    // VCGE.U8 d0, d0, d0
uint16x4_t vcle_u16(uint16x4_t a, uint16x4_t b); // VCGE.U16 d0, d0, d0
uint32x2_t vcle_u32(uint32x2_t a, uint32x2_t b); // VCGE.U32 d0, d0, d0
uint8x16_t vcleq_s8(int8x16_t a, int8x16_t b);  // VCGE.S8 q0, q0, q0
uint16x8_t vcleq_s16(int16x8_t a, int16x8_t b); // VCGE.S16 q0, q0, q0
uint32x4_t vcleq_s32(int32x4_t a, int32x4_t b); // VCGE.S32 q0, q0, q0
uint32x4_t vcleq_f32(float32x4_t a, float32x4_t b); // VCGE.F32 q0, q0, q0
uint8x16_t vcleq_u8(uint8x16_t a, uint8x16_t b); // VCGE.U8 q0, q0, q0
uint16x8_t vcleq_u16(uint16x8_t a, uint16x8_t b); // VCGE.U16 q0, q0, q0
uint32x4_t vcleq_u32(uint32x4_t a, uint32x4_t b); // VCGE.U32 q0, q0, q0

```

Vector compare greater-than

```

uint8x8_t  vcgt_s8(int8x8_t a, int8x8_t b);      // VCGT.S8 d0, d0, d0
uint16x4_t vcgt_s16(int16x4_t a, int16x4_t b);   // VCGT.S16 d0, d0, d0
uint32x2_t vcgt_s32(int32x2_t a, int32x2_t b);   // VCGT.S32 d0, d0, d0
uint32x2_t vcgt_f32(float32x2_t a, float32x2_t b); // VCGT.F32 d0, d0, d0
uint8x8_t  vcgt_u8(uint8x8_t a, uint8x8_t b);    // VCGT.U8 d0, d0, d0
uint16x4_t vcgt_u16(uint16x4_t a, uint16x4_t b); // VCGT.U16 d0, d0, d0
uint32x2_t vcgt_u32(uint32x2_t a, uint32x2_t b); // VCGT.U32 d0, d0, d0
uint8x16_t vcgtq_s8(int8x16_t a, int8x16_t b);   // VCGT.S8 q0, q0, q0
uint16x8_t vcgtq_s16(int16x8_t a, int16x8_t b);  // VCGT.S16 q0, q0, q0
uint32x4_t vcgtq_s32(int32x4_t a, int32x4_t b);  // VCGT.S32 q0, q0, q0
uint32x4_t vcgtq_f32(float32x4_t a, float32x4_t b); // VCGT.F32 q0, q0, q0
uint8x16_t vcgtq_u8(uint8x16_t a, uint8x16_t b); // VCGT.U8 q0, q0, q0
uint16x8_t vcgtq_u16(uint16x8_t a, uint16x8_t b); // VCGT.U16 q0, q0, q0
uint32x4_t vcgtq_u32(uint32x4_t a, uint32x4_t b); // VCGT.U32 q0, q0, q0

```

Vector compare less-than

```

uint8x8_t  vclt_s8(int8x8_t a, int8x8_t b);      // VCGT.S8 d0, d0, d0
uint16x4_t vclt_s16(int16x4_t a, int16x4_t b);   // VCGT.S16 d0, d0, d0
uint32x2_t vclt_s32(int32x2_t a, int32x2_t b);   // VCGT.S32 d0, d0, d0
uint32x2_t vclt_f32(float32x2_t a, float32x2_t b); // VCGT.F32 d0, d0, d0
uint8x8_t  vclt_u8(uint8x8_t a, uint8x8_t b);    // VCGT.U8 d0, d0, d0
uint16x4_t vclt_u16(uint16x4_t a, uint16x4_t b); // VCGT.U16 d0, d0, d0
uint32x2_t vclt_u32(uint32x2_t a, uint32x2_t b); // VCGT.U32 d0, d0, d0
uint8x16_t vcltq_s8(int8x16_t a, int8x16_t b);   // VCGT.S8 q0, q0, q0
uint16x8_t vcltq_s16(int16x8_t a, int16x8_t b);  // VCGT.S16 q0, q0, q0
uint32x4_t vcltq_s32(int32x4_t a, int32x4_t b);  // VCGT.S32 q0, q0, q0
uint32x4_t vcltq_f32(float32x4_t a, float32x4_t b); // VCGT.F32 q0, q0, q0
uint8x16_t vcltq_u8(uint8x16_t a, uint8x16_t b); // VCGT.U8 q0, q0, q0
uint16x8_t vcltq_u16(uint16x8_t a, uint16x8_t b); // VCGT.U16 q0, q0, q0
uint32x4_t vcltq_u32(uint32x4_t a, uint32x4_t b); // VCGT.U32 q0, q0, q0

```

Vector compare absolute greater-than or equal

```
uint32x2_t vcage_f32(float32x2_t a, float32x2_t b); // VACGE.F32 d0, d0, d0
uint32x4_t vcageq_f32(float32x4_t a, float32x4_t b); // VACGE.F32 q0, q0, q0
```

Vector compare absolute less-than or equal

```
uint32x2_t vcale_f32(float32x2_t a, float32x2_t b); // VACGE.F32 d0, d0, d0
uint32x4_t vcaleq_f32(float32x4_t a, float32x4_t b); // VACGE.F32 q0, q0, q0
```

Vector compare absolute greater-than

```
uint32x2_t vcagt_f32(float32x2_t a, float32x2_t b); // VACGT.F32 d0, d0, d0
uint32x4_t vcagtq_f32(float32x4_t a, float32x4_t b); // VACGT.F32 q0, q0, q0
```

Vector compare absolute less-than

```
uint32x2_t vcalt_f32(float32x2_t a, float32x2_t b); // VACGT.F32 d0, d0, d0
uint32x4_t vcaltq_f32(float32x4_t a, float32x4_t b); // VACGT.F32 q0, q0, q0
```

Vector test bits

```
uint8x8_t vtst_s8(int8x8_t a, int8x8_t b); // VTST.8 d0, d0, d0
uint16x4_t vtst_s16(int16x4_t a, int16x4_t b); // VTST.16 d0, d0, d0
uint32x2_t vtst_s32(int32x2_t a, int32x2_t b); // VTST.32 d0, d0, d0
uint8x8_t vtst_u8(uint8x8_t a, uint8x8_t b); // VTST.8 d0, d0, d0
uint16x4_t vtst_u16(uint16x4_t a, uint16x4_t b); // VTST.16 d0, d0, d0
uint32x2_t vtst_u32(uint32x2_t a, uint32x2_t b); // VTST.32 d0, d0, d0
uint8x8_t vtst_p8(poly8x8_t a, poly8x8_t b); // VTST.8 d0, d0, d0
uint8x16_t vtstq_s8(int8x16_t a, int8x16_t b); // VTST.8 q0, q0, q0
uint16x8_t vtstq_s16(int16x8_t a, int16x8_t b); // VTST.16 q0, q0, q0
uint32x4_t vtstq_s32(int32x4_t a, int32x4_t b); // VTST.32 q0, q0, q0
uint8x16_t vtstq_u8(uint8x16_t a, uint8x16_t b); // VTST.8 q0, q0, q0
uint16x8_t vtstq_u16(uint16x8_t a, uint16x8_t b); // VTST.16 q0, q0, q0
uint32x4_t vtstq_u32(uint32x4_t a, uint32x4_t b); // VTST.32 q0, q0, q0
uint8x16_t vtstq_p8(poly8x16_t a, poly8x16_t b); // VTST.8 q0, q0, q0
```

E.3.5 Absolute difference

These intrinsics provide operations including absolute difference.

Absolute difference between the arguments: $Vr[i] = |Va[i] - Vb[i]|$

```
int8x8_t vabd_s8(int8x8_t a, int8x8_t b); // VABD.S8 d0,d0,d0
int16x4_t vabd_s16(int16x4_t a, int16x4_t b); // VABD.S16 d0,d0,d0
int32x2_t vabd_s32(int32x2_t a, int32x2_t b); // VABD.S32 d0,d0,d0
uint8x8_t vabd_u8(uint8x8_t a, uint8x8_t b); // VABD.U8 d0,d0,d0
```

```

uint16x4_t vabd_u16(uint16x4_t a, uint16x4_t b);    // VABD.U16 d0,d0,d0
uint32x2_t vabd_u32(uint32x2_t a, uint32x2_t b);    // VABD.U32 d0,d0,d0
float32x2_t vabd_f32(float32x2_t a, float32x2_t b); // VABD.F32 d0,d0,d0
int8x16_t vabdq_s8(int8x16_t a, int8x16_t b);      // VABD.S8 q0,q0,q0
int16x8_t vabdq_s16(int16x8_t a, int16x8_t b);     // VABD.S16 q0,q0,q0
int32x4_t vabdq_s32(int32x4_t a, int32x4_t b);     // VABD.S32 q0,q0,q0
uint8x16_t vabdq_u8(uint8x16_t a, uint8x16_t b);   // VABD.U8 q0,q0,q0
uint16x8_t vabdq_u16(uint16x8_t a, uint16x8_t b);  // VABD.U16 q0,q0,q0
uint32x4_t vabdq_u32(uint32x4_t a, uint32x4_t b);  // VABD.U32 q0,q0,q0
float32x4_t vabdq_f32(float32x4_t a, float32x4_t b); // VABD.F32 q0,q0,q0

```

Absolute difference - long

```

int16x8_t vabdl_s8(int8x8_t a, int8x8_t b);        // VABDL.S8 q0,d0,d0
int32x4_t vabdl_s16(int16x4_t a, int16x4_t b);     // VABDL.S16 q0,d0,d0
int64x2_t vabdl_s32(int32x2_t a, int32x2_t b);     // VABDL.S32 q0,d0,d0
uint16x8_t vabdl_u8(uint8x8_t a, uint8x8_t b);     // VABDL.U8 q0,d0,d0
uint32x4_t vabdl_u16(uint16x4_t a, uint16x4_t b);  // VABDL.U16 q0,d0,d0
uint64x2_t vabdl_u32(uint32x2_t a, uint32x2_t b);  // VABDL.U32 q0,d0,d0

```

Absolute difference and accumulate: $Vr[i] = Va[i] + |Vb[i] - Vc[i]|$

```

int8x8_t vaba_s8(int8x8_t a, int8x8_t b, int8x8_t c); // VABA.S8 d0,d0,d0
int16x4_t vaba_s16(int16x4_t a, int16x4_t b, int16x4_t c); // VABA.S16 d0,d0,d0
int32x2_t vaba_s32(int32x2_t a, int32x2_t b, int32x2_t c); // VABA.S32 d0,d0,d0
uint8x8_t vaba_u8(uint8x8_t a, uint8x8_t b, uint8x8_t c); // VABA.U8 d0,d0,d0
uint16x4_t vaba_u16(uint16x4_t a, uint16x4_t b, uint16x4_t c); // VABA.U16 d0,d0,d0
uint32x2_t vaba_u32(uint32x2_t a, uint32x2_t b, uint32x2_t c); // VABA.U32 d0,d0,d0
int8x16_t vabaq_s8(int8x16_t a, int8x16_t b, int8x16_t c); // VABA.S8 q0,q0,q0
int16x8_t vabaq_s16(int16x8_t a, int16x8_t b, int16x8_t c); // VABA.S16 q0,q0,q0
int32x4_t vabaq_s32(int32x4_t a, int32x4_t b, int32x4_t c); // VABA.S32 q0,q0,q0
uint8x16_t vabaq_u8(uint8x16_t a, uint8x16_t b, uint8x16_t c); // VABA.U8 q0,q0,q0
uint16x8_t vabaq_u16(uint16x8_t a, uint16x8_t b, uint16x8_t c); // VABA.U16 q0,q0,q0
uint32x4_t vabaq_u32(uint32x4_t a, uint32x4_t b, uint32x4_t c); // VABA.U32 q0,q0,q0

```

Absolute difference and accumulate - long

```

int16x8_t vabal_s8(int16x8_t a, int8x8_t b, int8x8_t c); // VABAL.S8 q0,d0,d0
int32x4_t vabal_s16(int32x4_t a, int16x4_t b, int16x4_t c); // VABAL.S16 q0,d0,d0
int64x2_t vabal_s32(int64x2_t a, int32x2_t b, int32x2_t c); // VABAL.S32 q0,d0,d0
uint16x8_t vabal_u8(uint16x8_t a, uint8x8_t b, uint8x8_t c); // VABAL.U8 q0,d0,d0
uint32x4_t vabal_u16(uint32x4_t a, uint16x4_t b, uint16x4_t c); // VABAL.U16 q0,d0,d0
uint64x2_t vabal_u32(uint64x2_t a, uint32x2_t b, uint32x2_t c); // VABAL.U32 q0,d0,d0

```

E.3.6 Max/Min

These intrinsics provide maximum and minimum operations.

vmax -> Vr[i] := (Va[i] >= Vb[i]) ? Va[i] : Vb[i]

```

int8x8_t    vmax_s8(int8x8_t a, int8x8_t b);           // VMAX.S8 d0,d0,d0
int16x4_t   vmax_s16(int16x4_t a, int16x4_t b);        // VMAX.S16 d0,d0,d0
int32x2_t   vmax_s32(int32x2_t a, int32x2_t b);        // VMAX.S32 d0,d0,d0
uint8x8_t   vmax_u8(uint8x8_t a, uint8x8_t b);         // VMAX.U8 d0,d0,d0
uint16x4_t  vmax_u16(uint16x4_t a, uint16x4_t b);      // VMAX.U16 d0,d0,d0
uint32x2_t  vmax_u32(uint32x2_t a, uint32x2_t b);      // VMAX.U32 d0,d0,d0
float32x2_t vmax_f32(float32x2_t a, float32x2_t b);    // VMAX.F32 d0,d0,d0
int8x16_t   vmaxq_s8(int8x16_t a, int8x16_t b);        // VMAX.S8 q0,q0,q0
int16x8_t   vmaxq_s16(int16x8_t a, int16x8_t b);       // VMAX.S16 q0,q0,q0
int32x4_t   vmaxq_s32(int32x4_t a, int32x4_t b);       // VMAX.S32 q0,q0,q0
uint8x16_t  vmaxq_u8(uint8x16_t a, uint8x16_t b);      // VMAX.U8 q0,q0,q0
uint16x8_t  vmaxq_u16(uint16x8_t a, uint16x8_t b);     // VMAX.U16 q0,q0,q0
uint32x4_t  vmaxq_u32(uint32x4_t a, uint32x4_t b);     // VMAX.U32 q0,q0,q0
float32x4_t vmaxq_f32(float32x4_t a, float32x4_t b);   // VMAX.F32 q0,q0,q0

```

vmin -> Vr[i] := (Va[i] >= Vb[i]) ? Vb[i] : Va[i]

```

int8x8_t    vmin_s8(int8x8_t a, int8x8_t b);           // VMIN.S8 d0,d0,d0
int16x4_t   vmin_s16(int16x4_t a, int16x4_t b);        // VMIN.S16 d0,d0,d0
int32x2_t   vmin_s32(int32x2_t a, int32x2_t b);        // VMIN.S32 d0,d0,d0
uint8x8_t   vmin_u8(uint8x8_t a, uint8x8_t b);         // VMIN.U8 d0,d0,d0
uint16x4_t  vmin_u16(uint16x4_t a, uint16x4_t b);      // VMIN.U16 d0,d0,d0
uint32x2_t  vmin_u32(uint32x2_t a, uint32x2_t b);      // VMIN.U32 d0,d0,d0
float32x2_t vmin_f32(float32x2_t a, float32x2_t b);    // VMIN.F32 d0,d0,d0
int8x16_t   vminq_s8(int8x16_t a, int8x16_t b);        // VMIN.S8 q0,q0,q0
int16x8_t   vminq_s16(int16x8_t a, int16x8_t b);       // VMIN.S16 q0,q0,q0
int32x4_t   vminq_s32(int32x4_t a, int32x4_t b);       // VMIN.S32 q0,q0,q0
uint8x16_t  vminq_u8(uint8x16_t a, uint8x16_t b);      // VMIN.U8 q0,q0,q0
uint16x8_t  vminq_u16(uint16x8_t a, uint16x8_t b);     // VMIN.U16 q0,q0,q0
uint32x4_t  vminq_u32(uint32x4_t a, uint32x4_t b);     // VMIN.U32 q0,q0,q0
float32x4_t vminq_f32(float32x4_t a, float32x4_t b);   // VMIN.F32 q0,q0,q0

```

E.3.7 Pairwise addition

These intrinsics provide pairwise addition operations.

Pairwise add

```

int8x8_t    vpadd_s8(int8x8_t a, int8x8_t b);          // VPADD.I8 d0,d0,d0
int16x4_t   vpadd_s16(int16x4_t a, int16x4_t b);       // VPADD.I16 d0,d0,d0
int32x2_t   vpadd_s32(int32x2_t a, int32x2_t b);       // VPADD.I32 d0,d0,d0
uint8x8_t   vpadd_u8(uint8x8_t a, uint8x8_t b);        // VPADD.U8 d0,d0,d0
uint16x4_t  vpadd_u16(uint16x4_t a, uint16x4_t b);     // VPADD.U16 d0,d0,d0
uint32x2_t  vpadd_u32(uint32x2_t a, uint32x2_t b);     // VPADD.U32 d0,d0,d0
float32x2_t vpadd_f32(float32x2_t a, float32x2_t b);   // VPADD.F32 d0,d0,d0

```

Long pairwise add

```

int16x4_t  vpaddl_s8(int8x8_t a);           // VPADDL.S8 d0,d0
int32x2_t  vpaddl_s16(int16x4_t a);         // VPADDL.S16 d0,d0
int64x1_t  vpaddl_s32(int32x2_t a);         // VPADDL.S32 d0,d0
uint16x4_t vpaddl_u8(uint8x8_t a);          // VPADDL.U8 d0,d0
uint32x2_t vpaddl_u16(uint16x4_t a);        // VPADDL.U16 d0,d0
uint64x1_t vpaddl_u32(uint32x2_t a);        // VPADDL.U32 d0,d0
int16x8_t  vpaddlq_s8(int8x16_t a);          // VPADDL.S8 q0,q0
int32x4_t  vpaddlq_s16(int16x8_t a);         // VPADDL.S16 q0,q0
int64x2_t  vpaddlq_s32(int32x4_t a);         // VPADDL.S32 q0,q0
uint16x8_t vpaddlq_u8(uint8x16_t a);         // VPADDL.U8 q0,q0
uint32x4_t vpaddlq_u16(uint16x8_t a);        // VPADDL.U16 q0,q0
uint64x2_t vpaddlq_u32(uint32x4_t a);        // VPADDL.U32 q0,q0

```

Long pairwise add and accumulate

```

int16x4_t  vpadal_s8(int16x4_t a, int8x8_t b); // VPADAL.S8 d0,d0
int32x2_t  vpadal_s16(int32x2_t a, int16x4_t b); // VPADAL.S16 d0,d0
int64x1_t  vpadal_s32(int64x1_t a, int32x2_t b); // VPADAL.S32 d0,d0
uint16x4_t vpadal_u8(uint16x4_t a, uint8x8_t b); // VPADAL.U8 d0,d0
uint32x2_t vpadal_u16(uint32x2_t a, uint16x4_t b); // VPADAL.U16 d0,d0
uint64x1_t vpadal_u32(uint64x1_t a, uint32x2_t b); // VPADAL.U32 d0,d0
int16x8_t  vpadalq_s8(int16x8_t a, int8x16_t b); // VPADAL.S8 q0,q0
int32x4_t  vpadalq_s16(int32x4_t a, int16x8_t b); // VPADAL.S16 q0,q0
int64x2_t  vpadalq_s32(int64x2_t a, int32x4_t b); // VPADAL.S32 q0,q0
uint16x8_t vpadalq_u8(uint16x8_t a, uint8x16_t b); // VPADAL.U8 q0,q0
uint32x4_t vpadalq_u16(uint32x4_t a, uint16x8_t b); // VPADAL.U16 q0,q0
uint64x2_t vpadalq_u32(uint64x2_t a, uint32x4_t b); // VPADAL.U32 q0,q0

```

E.3.8 Folding maximum

vpmax -> takes maximum of adjacent pairs

```

int8x8_t    vpmax_s8(int8x8_t a, int8x8_t b); // VPMAX.S8 d0,d0,d0
int16x4_t    vpmax_s16(int16x4_t a, int16x4_t b); // VPMAX.S16 d0,d0,d0
int32x2_t    vpmax_s32(int32x2_t a, int32x2_t b); // VPMAX.S32 d0,d0,d0
uint8x8_t    vpmax_u8(uint8x8_t a, uint8x8_t b); // VPMAX.U8 d0,d0,d0
uint16x4_t    vpmax_u16(uint16x4_t a, uint16x4_t b); // VPMAX.U16 d0,d0,d0
uint32x2_t    vpmax_u32(uint32x2_t a, uint32x2_t b); // VPMAX.U32 d0,d0,d0
float32x2_t    vpmax_f32(float32x2_t a, float32x2_t b); // VPMAX.F32 d0,d0,d0

```

E.3.9 Folding minimum

vpmin -> takes minimum of adjacent pairs

```

int8x8_t    vpmin_s8(int8x8_t a, int8x8_t b); // VPMIN.S8 d0,d0,d0
int16x4_t    vpmin_s16(int16x4_t a, int16x4_t b); // VPMIN.S16 d0,d0,d0
int32x2_t    vpmin_s32(int32x2_t a, int32x2_t b); // VPMIN.S32 d0,d0,d0

```

```
uint8x8_t  vpmín_u8(uint8x8_t a, uint8x8_t b);    // VPMIN.U8 d0,d0,d0
uint16x4_t vpmín_u16(uint16x4_t a, uint16x4_t b); // VPMIN.U16 d0,d0,d0
uint32x2_t vpmín_u32(uint32x2_t a, uint32x2_t b); // VPMIN.U32 d0,d0,d0
float32x2_t vpmín_f32(float32x2_t a, float32x2_t b); // VPMIN.F32 d0,d0,d0
```

E.3.10 Reciprocal/Sqrt

Reciprocal estimate/step and 1/sqrt estimate/step

```
float32x2_t vrecps_f32(float32x2_t a, float32x2_t b); // VRECPS.F32 d0, d0, d0
float32x4_t vrecpsq_f32(float32x4_t a, float32x4_t b); // VRECPS.F32 q0, q0, q0
float32x2_t vrsqrts_f32(float32x2_t a, float32x2_t b); // VRSQRTS.F32 d0, d0, d0
float32x4_t vrsqrtsq_f32(float32x4_t a, float32x4_t b); // VRSQRTS.F32 q0, q0, q0
```

E.3.11 Shifts by signed variable

These intrinsics provide operations including shift by signed variable.

Vector shift left: $Vr[i] := Va[i] \ll Vb[i]$ (negative values shift right)

```
int8x8_t   vshl_s8(int8x8_t a, int8x8_t b);    // VSHL.S8 d0,d0,d0
int16x4_t  vshl_s16(int16x4_t a, int16x4_t b); // VSHL.S16 d0,d0,d0
int32x2_t  vshl_s32(int32x2_t a, int32x2_t b); // VSHL.S32 d0,d0,d0
int64x1_t  vshl_s64(int64x1_t a, int64x1_t b); // VSHL.S64 d0,d0,d0
uint8x8_t  vshl_u8(uint8x8_t a, int8x8_t b);   // VSHL.U8 d0,d0,d0
uint16x4_t vshl_u16(uint16x4_t a, int16x4_t b); // VSHL.U16 d0,d0,d0
uint32x2_t vshl_u32(uint32x2_t a, int32x2_t b); // VSHL.U32 d0,d0,d0
uint64x1_t vshl_u64(uint64x1_t a, int64x1_t b); // VSHL.U64 d0,d0,d0
int8x16_t  vshlq_s8(int8x16_t a, int8x16_t b); // VSHL.S8 q0,q0,q0
int16x8_t  vshlq_s16(int16x8_t a, int16x8_t b); // VSHL.S16 q0,q0,q0
int32x4_t  vshlq_s32(int32x4_t a, int32x4_t b); // VSHL.S32 q0,q0,q0
int64x2_t  vshlq_s64(int64x2_t a, int64x2_t b); // VSHL.S64 q0,q0,q0
uint8x16_t vshlq_u8(uint8x16_t a, int8x16_t b); // VSHL.U8 q0,q0,q0
uint16x8_t vshlq_u16(uint16x8_t a, int16x8_t b); // VSHL.U16 q0,q0,q0
uint32x4_t vshlq_u32(uint32x4_t a, int32x4_t b); // VSHL.U32 q0,q0,q0
uint64x2_t vshlq_u64(uint64x2_t a, int64x2_t b); // VSHL.U64 q0,q0,q0
```

Vector saturating shift left: (negative values shift right)

```
int8x8_t   vqshl_s8(int8x8_t a, int8x8_t b);    // VQSHL.S8 d0,d0,d0
int16x4_t  vqshl_s16(int16x4_t a, int16x4_t b); // VQSHL.S16 d0,d0,d0
int32x2_t  vqshl_s32(int32x2_t a, int32x2_t b); // VQSHL.S32 d0,d0,d0
int64x1_t  vqshl_s64(int64x1_t a, int64x1_t b); // VQSHL.S64 d0,d0,d0
uint8x8_t  vqshl_u8(uint8x8_t a, int8x8_t b);   // VQSHL.U8 d0,d0,d0
uint16x4_t vqshl_u16(uint16x4_t a, int16x4_t b); // VQSHL.U16 d0,d0,d0
uint32x2_t vqshl_u32(uint32x2_t a, int32x2_t b); // VQSHL.U32 d0,d0,d0
uint64x1_t vqshl_u64(uint64x1_t a, int64x1_t b); // VQSHL.U64 d0,d0,d0
int8x16_t  vqshlq_s8(int8x16_t a, int8x16_t b); // VQSHL.S8 q0,q0,q0
```

```

int16x8_t  vqshlq_s16(int16x8_t a, int16x8_t b); // VQSHL.S16 q0,q0,q0
int32x4_t  vqshlq_s32(int32x4_t a, int32x4_t b); // VQSHL.S32 q0,q0,q0
int64x2_t  vqshlq_s64(int64x2_t a, int64x2_t b); // VQSHL.S64 q0,q0,q0
uint8x16_t vqshlq_u8(uint8x16_t a, int8x16_t b); // VQSHL.U8 q0,q0,q0
uint16x8_t vqshlq_u16(uint16x8_t a, int16x8_t b); // VQSHL.U16 q0,q0,q0
uint32x4_t vqshlq_u32(uint32x4_t a, int32x4_t b); // VQSHL.U32 q0,q0,q0
uint64x2_t vqshlq_u64(uint64x2_t a, int64x2_t b); // VQSHL.U64 q0,q0,q0

```

Vector rounding shift left: (negative values shift right)

```

int8x8_t   vrshl_s8(int8x8_t a, int8x8_t b); // VRSHL.S8 d0,d0,d0
int16x4_t  vrshl_s16(int16x4_t a, int16x4_t b); // VRSHL.S16 d0,d0,d0
int32x2_t  vrshl_s32(int32x2_t a, int32x2_t b); // VRSHL.S32 d0,d0,d0
int64x1_t  vrshl_s64(int64x1_t a, int64x1_t b); // VRSHL.S64 d0,d0,d0
uint8x8_t  vrshl_u8(uint8x8_t a, int8x8_t b); // VRSHL.U8 d0,d0,d0
uint16x4_t vrshl_u16(uint16x4_t a, int16x4_t b); // VRSHL.U16 d0,d0,d0
uint32x2_t vrshl_u32(uint32x2_t a, int32x2_t b); // VRSHL.U32 d0,d0,d0
uint64x1_t vrshl_u64(uint64x1_t a, int64x1_t b); // VRSHL.U64 d0,d0,d0
int8x16_t  vrshlq_s8(int8x16_t a, int8x16_t b); // VRSHL.S8 q0,q0,q0
int16x8_t  vrshlq_s16(int16x8_t a, int16x8_t b); // VRSHL.S16 q0,q0,q0
int32x4_t  vrshlq_s32(int32x4_t a, int32x4_t b); // VRSHL.S32 q0,q0,q0
int64x2_t  vrshlq_s64(int64x2_t a, int64x2_t b); // VRSHL.S64 q0,q0,q0
uint8x16_t vrshlq_u8(uint8x16_t a, int8x16_t b); // VRSHL.U8 q0,q0,q0
uint16x8_t vrshlq_u16(uint16x8_t a, int16x8_t b); // VRSHL.U16 q0,q0,q0
uint32x4_t vrshlq_u32(uint32x4_t a, int32x4_t b); // VRSHL.U32 q0,q0,q0
uint64x2_t vrshlq_u64(uint64x2_t a, int64x2_t b); // VRSHL.U64 q0,q0,q0

```

Vector saturating rounding shift left: (negative values shift right)

```

int8x8_t   vqrshl_s8(int8x8_t a, int8x8_t b); // VQRSHL.S8 d0,d0,d0
int16x4_t  vqrshl_s16(int16x4_t a, int16x4_t b); // VQRSHL.S16 d0,d0,d0
int32x2_t  vqrshl_s32(int32x2_t a, int32x2_t b); // VQRSHL.S32 d0,d0,d0
int64x1_t  vqrshl_s64(int64x1_t a, int64x1_t b); // VQRSHL.S64 d0,d0,d0
uint8x8_t  vqrshl_u8(uint8x8_t a, int8x8_t b); // VQRSHL.U8 d0,d0,d0
uint16x4_t vqrshl_u16(uint16x4_t a, int16x4_t b); // VQRSHL.U16 d0,d0,d0
uint32x2_t vqrshl_u32(uint32x2_t a, int32x2_t b); // VQRSHL.U32 d0,d0,d0
uint64x1_t vqrshl_u64(uint64x1_t a, int64x1_t b); // VQRSHL.U64 d0,d0,d0
int8x16_t  vqrshlq_s8(int8x16_t a, int8x16_t b); // VQRSHL.S8 q0,q0,q0
int16x8_t  vqrshlq_s16(int16x8_t a, int16x8_t b); // VQRSHL.S16 q0,q0,q0
int32x4_t  vqrshlq_s32(int32x4_t a, int32x4_t b); // VQRSHL.S32 q0,q0,q0
int64x2_t  vqrshlq_s64(int64x2_t a, int64x2_t b); // VQRSHL.S64 q0,q0,q0
uint8x16_t vqrshlq_u8(uint8x16_t a, int8x16_t b); // VQRSHL.U8 q0,q0,q0
uint16x8_t vqrshlq_u16(uint16x8_t a, int16x8_t b); // VQRSHL.U16 q0,q0,q0
uint32x4_t vqrshlq_u32(uint32x4_t a, int32x4_t b); // VQRSHL.U32 q0,q0,q0
uint64x2_t vqrshlq_u64(uint64x2_t a, int64x2_t b); // VQRSHL.U64 q0,q0,q0

```

E.3.12 Shifts by a constant

These intrinsics provide operations for shifting by a constant.

Vector shift right by constant

```
int8x8_t  vshr_n_s8(int8x8_t a, __constrange(1,8) int b);    // VSHR.S8 d0,d0,#8
int16x4_t vshr_n_s16(int16x4_t a, __constrange(1,16) int b); // VSHR.S16 d0,d0,#16
int32x2_t vshr_n_s32(int32x2_t a, __constrange(1,32) int b); // VSHR.S32 d0,d0,#32
int64x1_t vshr_n_s64(int64x1_t a, __constrange(1,64) int b); // VSHR.S64 d0,d0,#64
uint8x8_t vshr_n_u8(uint8x8_t a, __constrange(1,8) int b);   // VSHR.U8 d0,d0,#8
uint16x4_t vshr_n_u16(uint16x4_t a, __constrange(1,16) int b); // VSHR.U16 d0,d0,#16
uint32x2_t vshr_n_u32(uint32x2_t a, __constrange(1,32) int b); // VSHR.U32 d0,d0,#32
uint64x1_t vshr_n_u64(uint64x1_t a, __constrange(1,64) int b); // VSHR.U64 d0,d0,#64
int8x16_t vshrq_n_s8(int8x16_t a, __constrange(1,8) int b);  // VSHR.S8 q0,q0,#8
int16x8_t vshrq_n_s16(int16x8_t a, __constrange(1,16) int b); // VSHR.S16 q0,q0,#16
int32x4_t vshrq_n_s32(int32x4_t a, __constrange(1,32) int b); // VSHR.S32 q0,q0,#32
int64x2_t vshrq_n_s64(int64x2_t a, __constrange(1,64) int b); // VSHR.S64 q0,q0,#64
uint8x16_t vshrq_n_u8(uint8x16_t a, __constrange(1,8) int b); // VSHR.U8 q0,q0,#8
uint16x8_t vshrq_n_u16(uint16x8_t a, __constrange(1,16) int b); // VSHR.U16 q0,q0,#16
uint32x4_t vshrq_n_u32(uint32x4_t a, __constrange(1,32) int b); // VSHR.U32 q0,q0,#32
uint64x2_t vshrq_n_u64(uint64x2_t a, __constrange(1,64) int b); // VSHR.U64 q0,q0,#64
```

Vector shift left by constant

```
int8x8_t  vshl_n_s8(int8x8_t a, __constrange(0,7) int b);    // VSHL.I8 d0,d0,#0
int16x4_t vshl_n_s16(int16x4_t a, __constrange(0,15) int b); // VSHL.I16 d0,d0,#0
int32x2_t vshl_n_s32(int32x2_t a, __constrange(0,31) int b); // VSHL.I32 d0,d0,#0
int64x1_t vshl_n_s64(int64x1_t a, __constrange(0,63) int b); // VSHL.I64 d0,d0,#0
uint8x8_t vshl_n_u8(uint8x8_t a, __constrange(0,7) int b);   // VSHL.I8 d0,d0,#0
uint16x4_t vshl_n_u16(uint16x4_t a, __constrange(0,15) int b); // VSHL.I16 d0,d0,#0
uint32x2_t vshl_n_u32(uint32x2_t a, __constrange(0,31) int b); // VSHL.I32 d0,d0,#0
uint64x1_t vshl_n_u64(uint64x1_t a, __constrange(0,63) int b); // VSHL.I64 d0,d0,#0
int8x16_t vshlq_n_s8(int8x16_t a, __constrange(0,7) int b);  // VSHL.I8 q0,q0,#0
int16x8_t vshlq_n_s16(int16x8_t a, __constrange(0,15) int b); // VSHL.I16 q0,q0,#0
int32x4_t vshlq_n_s32(int32x4_t a, __constrange(0,31) int b); // VSHL.I32 q0,q0,#0
int64x2_t vshlq_n_s64(int64x2_t a, __constrange(0,63) int b); // VSHL.I64 q0,q0,#0
uint8x16_t vshlq_n_u8(uint8x16_t a, __constrange(0,7) int b); // VSHL.I8 q0,q0,#0
uint16x8_t vshlq_n_u16(uint16x8_t a, __constrange(0,15) int b); // VSHL.I16 q0,q0,#0
uint32x4_t vshlq_n_u32(uint32x4_t a, __constrange(0,31) int b); // VSHL.I32 q0,q0,#0
uint64x2_t vshlq_n_u64(uint64x2_t a, __constrange(0,63) int b); // VSHL.I64 q0,q0,#0
```

Vector rounding shift right by constant

```
int8x8_t  vrshr_n_s8(int8x8_t a, __constrange(1,8) int b);    // VRSHR.S8 d0,d0,#8
int16x4_t vrshr_n_s16(int16x4_t a, __constrange(1,16) int b); // VRSHR.S16 d0,d0,#16
int32x2_t vrshr_n_s32(int32x2_t a, __constrange(1,32) int b); // VRSHR.S32 d0,d0,#32
int64x1_t vrshr_n_s64(int64x1_t a, __constrange(1,64) int b); // VRSHR.S64 d0,d0,#64
```

```

uint8x8_t vrshr_n_u8(uint8x8_t a, __constrange(1,8) int b); // VRSHR.U8 d0,d0,#8
uint16x4_t vrshr_n_u16(uint16x4_t a, __constrange(1,16) int b); // VRSHR.U16 d0,d0,#16
uint32x2_t vrshr_n_u32(uint32x2_t a, __constrange(1,32) int b); // VRSHR.U32 d0,d0,#32
uint64x1_t vrshr_n_u64(uint64x1_t a, __constrange(1,64) int b); // VRSHR.U64 d0,d0,#64
int8x16_t vrshrq_n_s8(int8x16_t a, __constrange(1,8) int b); // VRSHR.S8 q0,q0,#8
int16x8_t vrshrq_n_s16(int16x8_t a, __constrange(1,16) int b); // VRSHR.S16 q0,q0,#16
int32x4_t vrshrq_n_s32(int32x4_t a, __constrange(1,32) int b); // VRSHR.S32 q0,q0,#32
int64x2_t vrshrq_n_s64(int64x2_t a, __constrange(1,64) int b); // VRSHR.S64 q0,q0,#64
uint8x16_t vrshrq_n_u8(uint8x16_t a, __constrange(1,8) int b); // VRSHR.U8 q0,q0,#8
uint16x8_t vrshrq_n_u16(uint16x8_t a, __constrange(1,16) int b); // VRSHR.U16 q0,q0,#16
uint32x4_t vrshrq_n_u32(uint32x4_t a, __constrange(1,32) int b); // VRSHR.U32 q0,q0,#32
uint64x2_t vrshrq_n_u64(uint64x2_t a, __constrange(1,64) int b); // VRSHR.U64 q0,q0,#64

```

Vector shift right by constant and accumulate

```

int8x8_t vsra_n_s8(int8x8_t a, int8x8_t b, __constrange(1,8) int c); // VSRA.S8 d0,d0,#8
int16x4_t vsra_n_s16(int16x4_t a, int16x4_t b, __constrange(1,16) int c); // VSRA.S16 d0,d0,#16
int32x2_t vsra_n_s32(int32x2_t a, int32x2_t b, __constrange(1,32) int c); // VSRA.S32 d0,d0,#32
int64x1_t vsra_n_s64(int64x1_t a, int64x1_t b, __constrange(1,64) int c); // VSRA.S64 d0,d0,#64
uint8x8_t vsra_n_u8(uint8x8_t a, uint8x8_t b, __constrange(1,8) int c); // VSRA.U8 d0,d0,#8
uint16x4_t vsra_n_u16(uint16x4_t a, uint16x4_t b, __constrange(1,16) int c); // VSRA.U16 d0,d0,#16
uint32x2_t vsra_n_u32(uint32x2_t a, uint32x2_t b, __constrange(1,32) int c); // VSRA.U32 d0,d0,#32
uint64x1_t vsra_n_u64(uint64x1_t a, uint64x1_t b, __constrange(1,64) int c); // VSRA.U64 d0,d0,#64
int8x16_t vsraq_n_s8(int8x16_t a, int8x16_t b, __constrange(1,8) int c); // VSRA.S8 q0,q0,#8
int16x8_t vsraq_n_s16(int16x8_t a, int16x8_t b, __constrange(1,16) int c); // VSRA.S16 q0,q0,#16
int32x4_t vsraq_n_s32(int32x4_t a, int32x4_t b, __constrange(1,32) int c); // VSRA.S32 q0,q0,#32
int64x2_t vsraq_n_s64(int64x2_t a, int64x2_t b, __constrange(1,64) int c); // VSRA.S64 q0,q0,#64
uint8x16_t vsraq_n_u8(uint8x16_t a, uint8x16_t b, __constrange(1,8) int c); // VSRA.U8 q0,q0,#8
uint16x8_t vsraq_n_u16(uint16x8_t a, uint16x8_t b, __constrange(1,16) int c); // VSRA.U16 q0,q0,#16
uint32x4_t vsraq_n_u32(uint32x4_t a, uint32x4_t b, __constrange(1,32) int c); // VSRA.U32 q0,q0,#32
uint64x2_t vsraq_n_u64(uint64x2_t a, uint64x2_t b, __constrange(1,64) int c); // VSRA.U64 q0,q0,#64

```

Vector rounding shift right by constant and accumulate

```

int8x8_t vrsra_n_s8(int8x8_t a, int8x8_t b, __constrange(1,8) int c); // VRSRA.S8 d0,d0,#8
int16x4_t vrsra_n_s16(int16x4_t a, int16x4_t b, __constrange(1,16) int c); // VRSRA.S16 d0,d0,#16
int32x2_t vrsra_n_s32(int32x2_t a, int32x2_t b, __constrange(1,32) int c); // VRSRA.S32 d0,d0,#32
int64x1_t vrsra_n_s64(int64x1_t a, int64x1_t b, __constrange(1,64) int c); // VRSRA.S64 d0,d0,#64
uint8x8_t vrsra_n_u8(uint8x8_t a, uint8x8_t b, __constrange(1,8) int c); // VRSRA.U8 d0,d0,#8
uint16x4_t vrsra_n_u16(uint16x4_t a, uint16x4_t b, __constrange(1,16) int c); // VRSRA.U16 d0,d0,#16
uint32x2_t vrsra_n_u32(uint32x2_t a, uint32x2_t b, __constrange(1,32) int c); // VRSRA.U32 d0,d0,#32
uint64x1_t vrsra_n_u64(uint64x1_t a, uint64x1_t b, __constrange(1,64) int c); // VRSRA.U64 d0,d0,#64
int8x16_t vrsraq_n_s8(int8x16_t a, int8x16_t b, __constrange(1,8) int c); // VRSRA.S8 q0,q0,#8
int16x8_t vrsraq_n_s16(int16x8_t a, int16x8_t b, __constrange(1,16) int c); // VRSRA.S16 q0,q0,#16
int32x4_t vrsraq_n_s32(int32x4_t a, int32x4_t b, __constrange(1,32) int c); // VRSRA.S32 q0,q0,#32
int64x2_t vrsraq_n_s64(int64x2_t a, int64x2_t b, __constrange(1,64) int c); // VRSRA.S64 q0,q0,#64
uint8x16_t vrsraq_n_u8(uint8x16_t a, uint8x16_t b, __constrange(1,8) int c); // VRSRA.U8 q0,q0,#8
uint16x8_t vrsraq_n_u16(uint16x8_t a, uint16x8_t b, __constrange(1,16) int c); // VRSRA.U16 q0,q0,#16

```

```
uint32x4_t vrsraq_n_u32(uint32x4_t a, uint32x4_t b, __constrange(1,32) int c); // VRSRA.U32 q0,q0,#32
uint64x2_t vrsraq_n_u64(uint64x2_t a, uint64x2_t b, __constrange(1,64) int c); // VRSRA.U64 q0,q0,#64
```

Vector saturating shift left by constant

```
int8x8_t  vqshl_n_s8(int8x8_t a, __constrange(0,7) int b); // VQSHL.S8 d0,d0,#0
int16x4_t vqshl_n_s16(int16x4_t a, __constrange(0,15) int b); // VQSHL.S16 d0,d0,#0
int32x2_t vqshl_n_s32(int32x2_t a, __constrange(0,31) int b); // VQSHL.S32 d0,d0,#0
int64x1_t vqshl_n_s64(int64x1_t a, __constrange(0,63) int b); // VQSHL.S64 d0,d0,#0
uint8x8_t vqshl_n_u8(uint8x8_t a, __constrange(0,7) int b); // VQSHL.U8 d0,d0,#0
uint16x4_t vqshl_n_u16(uint16x4_t a, __constrange(0,15) int b); // VQSHL.U16 d0,d0,#0
uint32x2_t vqshl_n_u32(uint32x2_t a, __constrange(0,31) int b); // VQSHL.U32 d0,d0,#0
uint64x1_t vqshl_n_u64(uint64x1_t a, __constrange(0,63) int b); // VQSHL.U64 d0,d0,#0
int8x16_t vqshlq_n_s8(int8x16_t a, __constrange(0,7) int b); // VQSHL.S8 q0,q0,#0
int16x8_t vqshlq_n_s16(int16x8_t a, __constrange(0,15) int b); // VQSHL.S16 q0,q0,#0
int32x4_t vqshlq_n_s32(int32x4_t a, __constrange(0,31) int b); // VQSHL.S32 q0,q0,#0
int64x2_t vqshlq_n_s64(int64x2_t a, __constrange(0,63) int b); // VQSHL.S64 q0,q0,#0
uint8x16_t vqshlq_n_u8(uint8x16_t a, __constrange(0,7) int b); // VQSHL.U8 q0,q0,#0
uint16x8_t vqshlq_n_u16(uint16x8_t a, __constrange(0,15) int b); // VQSHL.U16 q0,q0,#0
uint32x4_t vqshlq_n_u32(uint32x4_t a, __constrange(0,31) int b); // VQSHL.U32 q0,q0,#0
uint64x2_t vqshlq_n_u64(uint64x2_t a, __constrange(0,63) int b); // VQSHL.U64 q0,q0,#0
```

Vector signed->unsigned saturating shift left by constant

```
uint8x8_t vqshlu_n_s8(int8x8_t a, __constrange(0,7) int b); // VQSHLU.S8 d0,d0,#0
uint16x4_t vqshlu_n_s16(int16x4_t a, __constrange(0,15) int b); // VQSHLU.S16 d0,d0,#0
uint32x2_t vqshlu_n_s32(int32x2_t a, __constrange(0,31) int b); // VQSHLU.S32 d0,d0,#0
uint64x1_t vqshlu_n_s64(int64x1_t a, __constrange(0,63) int b); // VQSHLU.S64 d0,d0,#0
uint8x16_t vqshluq_n_s8(int8x16_t a, __constrange(0,7) int b); // VQSHLU.S8 q0,q0,#0
uint16x8_t vqshluq_n_s16(int16x8_t a, __constrange(0,15) int b); // VQSHLU.S16 q0,q0,#0
uint32x4_t vqshluq_n_s32(int32x4_t a, __constrange(0,31) int b); // VQSHLU.S32 q0,q0,#0
uint64x2_t vqshluq_n_s64(int64x2_t a, __constrange(0,63) int b); // VQSHLU.S64 q0,q0,#0
```

Vector narrowing saturating shift right by constant

```
int8x8_t vshrn_n_s16(int16x8_t a, __constrange(1,8) int b); // VSHRN.I16 d0,q0,#8
int16x4_t vshrn_n_s32(int32x4_t a, __constrange(1,16) int b); // VSHRN.I32 d0,q0,#16
int32x2_t vshrn_n_s64(int64x2_t a, __constrange(1,32) int b); // VSHRN.I64 d0,q0,#32
uint8x8_t vshrn_n_u16(uint16x8_t a, __constrange(1,8) int b); // VSHRN.U16 d0,q0,#8
uint16x4_t vshrn_n_u32(uint32x4_t a, __constrange(1,16) int b); // VSHRN.U32 d0,q0,#16
uint32x2_t vshrn_n_u64(uint64x2_t a, __constrange(1,32) int b); // VSHRN.U64 d0,q0,#32
```

Vector signed->unsigned narrowing saturating shift right by constant

```
uint8x8_t vqshrun_n_s16(int16x8_t a, __constrange(1,8) int b); // VQSHRUN.S16 d0,q0,#8
uint16x4_t vqshrun_n_s32(int32x4_t a, __constrange(1,16) int b); // VQSHRUN.S32 d0,q0,#16
uint32x2_t vqshrun_n_s64(int64x2_t a, __constrange(1,32) int b); // VQSHRUN.S64 d0,q0,#32
```

Vector signed->unsigned rounding narrowing saturating shift right by constant

```
uint8x8_t  vqrshrun_n_s16(int16x8_t a, __constrange(1,8) int b); // VQRSHRUN.S16 d0,q0,#8
uint16x4_t vqrshrun_n_s32(int32x4_t a, __constrange(1,16) int b); // VQRSHRUN.S32 d0,q0,#16
uint32x2_t vqrshrun_n_s64(int64x2_t a, __constrange(1,32) int b); // VQRSHRUN.S64 d0,q0,#32
```

Vector narrowing saturating shift right by constant

```
int8x8_t   vqshrn_n_s16(int16x8_t a, __constrange(1,8) int b); // VQSHRN.S16 d0,q0,#8
int16x4_t  vqshrn_n_s32(int32x4_t a, __constrange(1,16) int b); // VQSHRN.S32 d0,q0,#16
int32x2_t  vqshrn_n_s64(int64x2_t a, __constrange(1,32) int b); // VQSHRN.S64 d0,q0,#32
uint8x8_t  vqshrn_n_u16(uint16x8_t a, __constrange(1,8) int b); // VQSHRN.U16 d0,q0,#8
uint16x4_t vqshrn_n_u32(uint32x4_t a, __constrange(1,16) int b); // VQSHRN.U32 d0,q0,#16
uint32x2_t vqshrn_n_u64(uint64x2_t a, __constrange(1,32) int b); // VQSHRN.U64 d0,q0,#32
```

Vector rounding narrowing shift right by constant

```
int8x8_t   vrshrn_n_s16(int16x8_t a, __constrange(1,8) int b); // VRSHRN.I16 d0,q0,#8
int16x4_t  vrshrn_n_s32(int32x4_t a, __constrange(1,16) int b); // VRSHRN.I32 d0,q0,#16
int32x2_t  vrshrn_n_s64(int64x2_t a, __constrange(1,32) int b); // VRSHRN.I64 d0,q0,#32
uint8x8_t  vrshrn_n_u16(uint16x8_t a, __constrange(1,8) int b); // VRSHRN.I16 d0,q0,#8
uint16x4_t vrshrn_n_u32(uint32x4_t a, __constrange(1,16) int b); // VRSHRN.I32 d0,q0,#16
uint32x2_t vrshrn_n_u64(uint64x2_t a, __constrange(1,32) int b); // VRSHRN.I64 d0,q0,#32
```

Vector rounding narrowing saturating shift right by constant

```
int8x8_t   vqrshrn_n_s16(int16x8_t a, __constrange(1,8) int b); // VQRSHRN.S16 d0,q0,#8
int16x4_t  vqrshrn_n_s32(int32x4_t a, __constrange(1,16) int b); // VQRSHRN.S32 d0,q0,#16
int32x2_t  vqrshrn_n_s64(int64x2_t a, __constrange(1,32) int b); // VQRSHRN.S64 d0,q0,#32
uint8x8_t  vqrshrn_n_u16(uint16x8_t a, __constrange(1,8) int b); // VQRSHRN.U16 d0,q0,#8
uint16x4_t vqrshrn_n_u32(uint32x4_t a, __constrange(1,16) int b); // VQRSHRN.U32 d0,q0,#16
uint32x2_t vqrshrn_n_u64(uint64x2_t a, __constrange(1,32) int b); // VQRSHRN.U64 d0,q0,#32
```

Vector widening shift left by constant

```
int16x8_t  vshll_n_s8(int8x8_t a, __constrange(0,7) int b); // VSHLL.S8 q0,d0,#0
int32x4_t  vshll_n_s16(int16x4_t a, __constrange(0,15) int b); // VSHLL.S16 q0,d0,#0
int64x2_t  vshll_n_s32(int32x2_t a, __constrange(0,31) int b); // VSHLL.S32 q0,d0,#0
uint16x8_t vshll_n_u8(uint8x8_t a, __constrange(0,7) int b); // VSHLL.U8 q0,d0,#0
uint32x4_t vshll_n_u16(uint16x4_t a, __constrange(0,15) int b); // VSHLL.U16 q0,d0,#0
uint64x2_t vshll_n_u32(uint32x2_t a, __constrange(0,31) int b); // VSHLL.U32 q0,d0,#0
```

E.3.13 Shifts with insert

These intrinsics provide operations including shifts with insert.

Vector shift right and insert

```

int8x8_t  vsri_n_s8(int8x8_t a, int8x8_t b, __constrange(1,8) int c);    // VSRI.8 d0,d0,#8
int16x4_t vsri_n_s16(int16x4_t a, int16x4_t b, __constrange(1,16) int c); // VSRI.16 d0,d0,#16
int32x2_t vsri_n_s32(int32x2_t a, int32x2_t b, __constrange(1,32) int c); // VSRI.32 d0,d0,#32
int64x1_t vsri_n_s64(int64x1_t a, int64x1_t b, __constrange(1,64) int c); // VSRI.64 d0,d0,#64
uint8x8_t vsri_n_u8(uint8x8_t a, uint8x8_t b, __constrange(1,8) int c);    // VSRI.8 d0,d0,#8
uint16x4_t vsri_n_u16(uint16x4_t a, uint16x4_t b, __constrange(1,16) int c); // VSRI.16 d0,d0,#16
uint32x2_t vsri_n_u32(uint32x2_t a, uint32x2_t b, __constrange(1,32) int c); // VSRI.32 d0,d0,#32
uint64x1_t vsri_n_u64(uint64x1_t a, uint64x1_t b, __constrange(1,64) int c); // VSRI.64 d0,d0,#64
poly8x8_t vsri_n_p8(poly8x8_t a, poly8x8_t b, __constrange(1,8) int c);    // VSRI.8 d0,d0,#8
poly16x4_t vsri_n_p16(poly16x4_t a, poly16x4_t b, __constrange(1,16) int c); // VSRI.16 d0,d0,#16
int8x16_t vsriq_n_s8(int8x16_t a, int8x16_t b, __constrange(1,8) int c);    // VSRI.8 q0,q0,#8
int16x8_t vsriq_n_s16(int16x8_t a, int16x8_t b, __constrange(1,16) int c);    // VSRI.16 q0,q0,#16
int32x4_t vsriq_n_s32(int32x4_t a, int32x4_t b, __constrange(1,32) int c);    // VSRI.32 q0,q0,#32
int64x2_t vsriq_n_s64(int64x2_t a, int64x2_t b, __constrange(1,64) int c);    // VSRI.64 q0,q0,#64
uint8x16_t vsriq_n_u8(uint8x16_t a, uint8x16_t b, __constrange(1,8) int c);    // VSRI.8 q0,q0,#8
uint16x8_t vsriq_n_u16(uint16x8_t a, uint16x8_t b, __constrange(1,16) int c); // VSRI.16 q0,q0,#16
uint32x4_t vsriq_n_u32(uint32x4_t a, uint32x4_t b, __constrange(1,32) int c); // VSRI.32 q0,q0,#32
uint64x2_t vsriq_n_u64(uint64x2_t a, uint64x2_t b, __constrange(1,64) int c); // VSRI.64 q0,q0,#64
poly8x16_t vsriq_n_p8(poly8x16_t a, poly8x16_t b, __constrange(1,8) int c);    // VSRI.8 q0,q0,#8
poly16x8_t vsriq_n_p16(poly16x8_t a, poly16x8_t b, __constrange(1,16) int c); // VSRI.16 q0,q0,#16

```

Vector shift left and insert

```

int8x8_t  vsli_n_s8(int8x8_t a, int8x8_t b, __constrange(0,7) int c);    // VSLI.8 d0,d0,#0
int16x4_t vsli_n_s16(int16x4_t a, int16x4_t b, __constrange(0,15) int c); // VSLI.16 d0,d0,#0
int32x2_t vsli_n_s32(int32x2_t a, int32x2_t b, __constrange(0,31) int c); // VSLI.32 d0,d0,#0
int64x1_t vsli_n_s64(int64x1_t a, int64x1_t b, __constrange(0,63) int c); // VSLI.64 d0,d0,#0
uint8x8_t vsli_n_u8(uint8x8_t a, uint8x8_t b, __constrange(0,7) int c);    // VSLI.8 d0,d0,#0
uint16x4_t vsli_n_u16(uint16x4_t a, uint16x4_t b, __constrange(0,15) int c); // VSLI.16 d0,d0,#0
uint32x2_t vsli_n_u32(uint32x2_t a, uint32x2_t b, __constrange(0,31) int c); // VSLI.32 d0,d0,#0
uint64x1_t vsli_n_u64(uint64x1_t a, uint64x1_t b, __constrange(0,63) int c); // VSLI.64 d0,d0,#0
poly8x8_t vsli_n_p8(poly8x8_t a, poly8x8_t b, __constrange(0,7) int c);    // VSLI.8 d0,d0,#0
poly16x4_t vsli_n_p16(poly16x4_t a, poly16x4_t b, __constrange(0,15) int c); // VSLI.16 d0,d0,#0
int8x16_t vsliq_n_s8(int8x16_t a, int8x16_t b, __constrange(0,7) int c);    // VSLI.8 q0,q0,#0
int16x8_t vsliq_n_s16(int16x8_t a, int16x8_t b, __constrange(0,15) int c);    // VSLI.16 q0,q0,#0
int32x4_t vsliq_n_s32(int32x4_t a, int32x4_t b, __constrange(0,31) int c);    // VSLI.32 q0,q0,#0
int64x2_t vsliq_n_s64(int64x2_t a, int64x2_t b, __constrange(0,63) int c);    // VSLI.64 q0,q0,#0
uint8x16_t vsliq_n_u8(uint8x16_t a, uint8x16_t b, __constrange(0,7) int c);    // VSLI.8 q0,q0,#0
uint16x8_t vsliq_n_u16(uint16x8_t a, uint16x8_t b, __constrange(0,15) int c); // VSLI.16 q0,q0,#0
uint32x4_t vsliq_n_u32(uint32x4_t a, uint32x4_t b, __constrange(0,31) int c); // VSLI.32 q0,q0,#0
uint64x2_t vsliq_n_u64(uint64x2_t a, uint64x2_t b, __constrange(0,63) int c); // VSLI.64 q0,q0,#0
poly8x16_t vsliq_n_p8(poly8x16_t a, poly8x16_t b, __constrange(0,7) int c);    // VSLI.8 q0,q0,#0
poly16x8_t vsliq_n_p16(poly16x8_t a, poly16x8_t b, __constrange(0,15) int c); // VSLI.16 q0,q0,#0

```

E.3.14 Loads and stores of a single vector

Perform loads and stores of a single vector of some type.

```

uint8x16_t vld1q_u8(__transfersize(16) uint8_t const * ptr);
// VLD1.8 {d0, d1}, [r0]
uint16x8_t vld1q_u16(__transfersize(8) uint16_t const * ptr);
// VLD1.16 {d0, d1}, [r0]
uint32x4_t vld1q_u32(__transfersize(4) uint32_t const * ptr);
// VLD1.32 {d0, d1}, [r0]
uint64x2_t vld1q_u64(__transfersize(2) uint64_t const * ptr);
// VLD1.64 {d0, d1}, [r0]
int8x16_t vld1q_s8(__transfersize(16) int8_t const * ptr);
// VLD1.8 {d0, d1}, [r0]
int16x8_t vld1q_s16(__transfersize(8) int16_t const * ptr);
// VLD1.16 {d0, d1}, [r0]
int32x4_t vld1q_s32(__transfersize(4) int32_t const * ptr);
// VLD1.32 {d0, d1}, [r0]
int64x2_t vld1q_s64(__transfersize(2) int64_t const * ptr);
// VLD1.64 {d0, d1}, [r0]
float32x4_t vld1q_f32(__transfersize(4) float32_t const * ptr);
// VLD1.32 {d0, d1}, [r0]
poly8x16_t vld1q_p8(__transfersize(16) poly8_t const * ptr);
// VLD1.8 {d0, d1}, [r0]
poly16x8_t vld1q_p16(__transfersize(8) poly16_t const * ptr);
// VLD1.16 {d0, d1}, [r0]
uint8x8_t vld1_u8(__transfersize(8) uint8_t const * ptr);
// VLD1.8 {d0}, [r0]
uint16x4_t vld1_u16(__transfersize(4) uint16_t const * ptr);
// VLD1.16 {d0}, [r0]
uint32x2_t vld1_u32(__transfersize(2) uint32_t const * ptr);
// VLD1.32 {d0}, [r0]
uint64x1_t vld1_u64(__transfersize(1) uint64_t const * ptr);
// VLD1.64 {d0}, [r0]
int8x8_t vld1_s8(__transfersize(8) int8_t const * ptr);
// VLD1.8 {d0}, [r0]
int16x4_t vld1_s16(__transfersize(4) int16_t const * ptr);
// VLD1.16 {d0}, [r0]
int32x2_t vld1_s32(__transfersize(2) int32_t const * ptr);
// VLD1.32 {d0}, [r0]
int64x1_t vld1_s64(__transfersize(1) int64_t const * ptr);
// VLD1.64 {d0}, [r0]
float32x2_t vld1_f32(__transfersize(2) float32_t const * ptr);
// VLD1.32 {d0}, [r0]
poly8x8_t vld1_p8(__transfersize(8) poly8_t const * ptr);
// VLD1.8 {d0}, [r0]
poly16x4_t vld1_p16(__transfersize(4) poly16_t const * ptr);
// VLD1.16 {d0}, [r0]
uint8x16_t vld1q_lane_u8(__transfersize(1) uint8_t const * ptr, uint8x16_t vec, __constrange(0,15) int
lane);
// VLD1.8 {d0[0]}, [r0]
uint16x8_t vld1q_lane_u16(__transfersize(1) uint16_t const * ptr, uint16x8_t vec, __constrange(0,7)
int lane);
// VLD1.16 {d0[0]}, [r0]

```

```

uint32x4_t vld1q_lane_u32(__transfersize(1) uint32_t const * ptr, uint32x4_t vec, __constrange(0,3)
int lane);
                                // VLD1.32 {d0[0]}, [r0]
uint64x2_t vld1q_lane_u64(__transfersize(1) uint64_t const * ptr, uint64x2_t vec, __constrange(0,1)
int lane);
                                // VLD1.64 {d0}, [r0]
int8x16_t vld1q_lane_s8(__transfersize(1) int8_t const * ptr, int8x16_t vec, __constrange(0,15) int
lane);
                                // VLD1.8 {d0[0]}, [r0]
int16x8_t vld1q_lane_s16(__transfersize(1) int16_t const * ptr, int16x8_t vec, __constrange(0,7) int
lane);
                                // VLD1.16 {d0[0]}, [r0]
int32x4_t vld1q_lane_s32(__transfersize(1) int32_t const * ptr, int32x4_t vec, __constrange(0,3) int
lane);
                                // VLD1.32 {d0[0]}, [r0]
float32x4_t vld1q_lane_f32(__transfersize(1) float32_t const * ptr, float32x4_t vec, __constrange(0,3)
int lane);
                                // VLD1.32 {d0[0]}, [r0]
int64x2_t vld1q_lane_s64(__transfersize(1) int64_t const * ptr, int64x2_t vec, __constrange(0,1) int
lane);
                                // VLD1.64 {d0}, [r0]
poly8x16_t vld1q_lane_p8(__transfersize(1) poly8_t const * ptr, poly8x16_t vec, __constrange(0,15) int
lane);
                                // VLD1.8 {d0[0]}, [r0]
poly16x8_t vld1q_lane_p16(__transfersize(1) poly16_t const * ptr, poly16x8_t vec, __constrange(0,7)
int lane);
                                // VLD1.16 {d0[0]}, [r0]
uint8x8_t vld1_lane_u8(__transfersize(1) uint8_t const * ptr, uint8x8_t vec, __constrange(0,7) int
lane);
                                // VLD1.8 {d0[0]}, [r0]
uint16x4_t vld1_lane_u16(__transfersize(1) uint16_t const * ptr, uint16x4_t vec, __constrange(0,3) int
lane);
                                // VLD1.16 {d0[0]}, [r0]
uint32x2_t vld1_lane_u32(__transfersize(1) uint32_t const * ptr, uint32x2_t vec, __constrange(0,1) int
lane);
                                // VLD1.32 {d0[0]}, [r0]
uint64x1_t vld1_lane_u64(__transfersize(1) uint64_t const * ptr, uint64x1_t vec, __constrange(0,0) int
lane);
                                // VLD1.64 {d0}, [r0]
int8x8_t vld1_lane_s8(__transfersize(1) int8_t const * ptr, int8x8_t vec, __constrange(0,7) int lane);
                                // VLD1.8 {d0[0]}, [r0]
int16x4_t vld1_lane_s16(__transfersize(1) int16_t const * ptr, int16x4_t vec, __constrange(0,3) int
lane);
                                // VLD1.16 {d0[0]}, [r0]
int32x2_t vld1_lane_s32(__transfersize(1) int32_t const * ptr, int32x2_t vec, __constrange(0,1) int
lane);
                                // VLD1.32 {d0[0]}, [r0]
float32x2_t vld1_lane_f32(__transfersize(1) float32_t const * ptr, float32x2_t vec, __constrange(0,1)
int lane);
                                // VLD1.32 {d0[0]}, [r0]

```

```

int64x1_t  vld1_lane_s64(__transfersize(1) int64_t const * ptr, int64x1_t vec, __constrange(0,0) int
lane);
// VLD1.64 {d0}, [r0]
poly8x8_t  vld1_lane_p8(__transfersize(1) poly8_t const * ptr, poly8x8_t vec, __constrange(0,7) int
lane);
// VLD1.8 {d0[0]}, [r0]
poly16x4_t vld1_lane_p16(__transfersize(1) poly16_t const * ptr, poly16x4_t vec, __constrange(0,3) int
lane);
// VLD1.16 {d0[0]}, [r0]
uint8x16_t vld1q_dup_u8(__transfersize(1) uint8_t const * ptr);
// VLD1.8 {d0[]}, [r0]
uint16x8_t vld1q_dup_u16(__transfersize(1) uint16_t const * ptr);
// VLD1.16 {d0[]}, [r0]
uint32x4_t vld1q_dup_u32(__transfersize(1) uint32_t const * ptr);
// VLD1.32 {d0[]}, [r0]
uint64x2_t vld1q_dup_u64(__transfersize(1) uint64_t const * ptr);
// VLD1.64 {d0}, [r0]
int8x16_t  vld1q_dup_s8(__transfersize(1) int8_t const * ptr);
// VLD1.8 {d0[]}, [r0]
int16x8_t  vld1q_dup_s16(__transfersize(1) int16_t const * ptr);
// VLD1.16 {d0[]}, [r0]
int32x4_t  vld1q_dup_s32(__transfersize(1) int32_t const * ptr);
// VLD1.32 {d0[]}, [r0]
int64x2_t  vld1q_dup_s64(__transfersize(1) int64_t const * ptr);
// VLD1.64 {d0}, [r0]
float32x4_t vld1q_dup_f32(__transfersize(1) float32_t const * ptr);
// VLD1.32 {d0[]}, [r0]
poly8x16_t vld1q_dup_p8(__transfersize(1) poly8_t const * ptr);
// VLD1.8 {d0[]}, [r0]
poly16x8_t vld1q_dup_p16(__transfersize(1) poly16_t const * ptr);
// VLD1.16 {d0[]}, [r0]
uint8x8_t  vld1_dup_u8(__transfersize(1) uint8_t const * ptr);
// VLD1.8 {d0[]}, [r0]
uint16x4_t vld1_dup_u16(__transfersize(1) uint16_t const * ptr);
// VLD1.16 {d0[]}, [r0]
uint32x2_t vld1_dup_u32(__transfersize(1) uint32_t const * ptr);
// VLD1.32 {d0[]}, [r0]
uint64x1_t vld1_dup_u64(__transfersize(1) uint64_t const * ptr);
// VLD1.64 {d0}, [r0]
int8x8_t  vld1_dup_s8(__transfersize(1) int8_t const * ptr);
// VLD1.8 {d0[]}, [r0]
int16x4_t  vld1_dup_s16(__transfersize(1) int16_t const * ptr);
// VLD1.16 {d0[]}, [r0]
int32x2_t  vld1_dup_s32(__transfersize(1) int32_t const * ptr);
// VLD1.32 {d0[]}, [r0]
int64x1_t  vld1_dup_s64(__transfersize(1) int64_t const * ptr);
// VLD1.64 {d0}, [r0]
float32x2_t vld1_dup_f32(__transfersize(1) float32_t const * ptr);
// VLD1.32 {d0[]}, [r0]
poly8x8_t  vld1_dup_p8(__transfersize(1) poly8_t const * ptr);

```

```

// VLD1.8 {d0[]}, [r0]
poly16x4_t vld1_dup_p16(__transfersize(1) poly16_t const * ptr);
// VLD1.16 {d0[]}, [r0]
void vst1q_u8(__transfersize(16) uint8_t * ptr, uint8x16_t val);
// VST1.8 {d0, d1}, [r0]
void vst1q_u16(__transfersize(8) uint16_t * ptr, uint16x8_t val);
// VST1.16 {d0, d1}, [r0]
void vst1q_u32(__transfersize(4) uint32_t * ptr, uint32x4_t val);
// VST1.32 {d0, d1}, [r0]
void vst1q_u64(__transfersize(2) uint64_t * ptr, uint64x2_t val);
// VST1.64 {d0, d1}, [r0]
void vst1q_s8(__transfersize(16) int8_t * ptr, int8x16_t val);
// VST1.8 {d0, d1}, [r0]
void vst1q_s16(__transfersize(8) int16_t * ptr, int16x8_t val);
// VST1.16 {d0, d1}, [r0]
void vst1q_s32(__transfersize(4) int32_t * ptr, int32x4_t val);
// VST1.32 {d0, d1}, [r0]
void vst1q_s64(__transfersize(2) int64_t * ptr, int64x2_t val);
// VST1.64 {d0, d1}, [r0]
void vst1q_f32(__transfersize(4) float32_t * ptr, float32x4_t val);
// VST1.32 {d0, d1}, [r0]
void vst1q_p8(__transfersize(16) poly8_t * ptr, poly8x16_t val);
// VST1.8 {d0, d1}, [r0]
void vst1q_p16(__transfersize(8) poly16_t * ptr, poly16x8_t val);
// VST1.16 {d0, d1}, [r0]
void vst1_u8(__transfersize(8) uint8_t * ptr, uint8x8_t val);
// VST1.8 {d0}, [r0]
void vst1_u16(__transfersize(4) uint16_t * ptr, uint16x4_t val);
// VST1.16 {d0}, [r0]
void vst1_u32(__transfersize(2) uint32_t * ptr, uint32x2_t val);
// VST1.32 {d0}, [r0]
void vst1_u64(__transfersize(1) uint64_t * ptr, uint64x1_t val);
// VST1.64 {d0}, [r0]
void vst1_s8(__transfersize(8) int8_t * ptr, int8x8_t val);
// VST1.8 {d0}, [r0]
void vst1_s16(__transfersize(4) int16_t * ptr, int16x4_t val);
// VST1.16 {d0}, [r0]
void vst1_s32(__transfersize(2) int32_t * ptr, int32x2_t val);
// VST1.32 {d0}, [r0]
void vst1_s64(__transfersize(1) int64_t * ptr, int64x1_t val);
// VST1.64 {d0}, [r0]
void vst1_f32(__transfersize(2) float32_t * ptr, float32x2_t val);
// VST1.32 {d0}, [r0]
void vst1_p8(__transfersize(8) poly8_t * ptr, poly8x8_t val);
// VST1.8 {d0}, [r0]
void vst1_p16(__transfersize(4) poly16_t * ptr, poly16x4_t val);
// VST1.16 {d0}, [r0]
void vst1q_lane_u8(__transfersize(1) uint8_t * ptr, uint8x16_t val, __constrange(0,15) int lane);
// VST1.8 {d0[0]}, [r0]
void vst1q_lane_u16(__transfersize(1) uint16_t * ptr, uint16x8_t val, __constrange(0,7) int lane);

```

```

// VST1.16 {d0[0]}, [r0]
void vst1q_lane_u32(__transfersize(1) uint32_t * ptr, uint32x4_t val, __constrange(0,3) int lane);
// VST1.32 {d0[0]}, [r0]
void vst1q_lane_u64(__transfersize(1) uint64_t * ptr, uint64x2_t val, __constrange(0,1) int lane);
// VST1.64 {d0}, [r0]
void vst1q_lane_s8(__transfersize(1) int8_t * ptr, int8x16_t val, __constrange(0,15) int lane);
// VST1.8 {d0[0]}, [r0]
void vst1q_lane_s16(__transfersize(1) int16_t * ptr, int16x8_t val, __constrange(0,7) int lane);
// VST1.16 {d0[0]}, [r0]
void vst1q_lane_s32(__transfersize(1) int32_t * ptr, int32x4_t val, __constrange(0,3) int lane);
// VST1.32 {d0[0]}, [r0]
void vst1q_lane_s64(__transfersize(1) int64_t * ptr, int64x2_t val, __constrange(0,1) int lane);
// VST1.64 {d0}, [r0]
void vst1q_lane_f32(__transfersize(1) float32_t * ptr, float32x4_t val, __constrange(0,3) int lane);
// VST1.32 {d0[0]}, [r0]
void vst1q_lane_p8(__transfersize(1) poly8_t * ptr, poly8x16_t val, __constrange(0,15) int lane);
// VST1.8 {d0[0]}, [r0]
void vst1q_lane_p16(__transfersize(1) poly16_t * ptr, poly16x8_t val, __constrange(0,7) int lane);
// VST1.16 {d0[0]}, [r0]
void vst1_lane_u8(__transfersize(1) uint8_t * ptr, uint8x8_t val, __constrange(0,7) int lane);
// VST1.8 {d0[0]}, [r0]
void vst1_lane_u16(__transfersize(1) uint16_t * ptr, uint16x4_t val, __constrange(0,3) int lane);
// VST1.16 {d0[0]}, [r0]
void vst1_lane_u32(__transfersize(1) uint32_t * ptr, uint32x2_t val, __constrange(0,1) int lane);
// VST1.32 {d0[0]}, [r0]
void vst1_lane_u64(__transfersize(1) uint64_t * ptr, uint64x1_t val, __constrange(0,0) int lane);
// VST1.64 {d0}, [r0]
void vst1_lane_s8(__transfersize(1) int8_t * ptr, int8x8_t val, __constrange(0,7) int lane);
// VST1.8 {d0[0]}, [r0]
void vst1_lane_s16(__transfersize(1) int16_t * ptr, int16x4_t val, __constrange(0,3) int lane);
// VST1.16 {d0[0]}, [r0]
void vst1_lane_s32(__transfersize(1) int32_t * ptr, int32x2_t val, __constrange(0,1) int lane);
// VST1.32 {d0[0]}, [r0]
void vst1_lane_s64(__transfersize(1) int64_t * ptr, int64x1_t val, __constrange(0,0) int lane);
// VST1.64 {d0}, [r0]
void vst1_lane_f32(__transfersize(1) float32_t * ptr, float32x2_t val, __constrange(0,1) int lane);
// VST1.32 {d0[0]}, [r0]
void vst1_lane_p8(__transfersize(1) poly8_t * ptr, poly8x8_t val, __constrange(0,7) int lane);
// VST1.8 {d0[0]}, [r0]
void vst1_lane_p16(__transfersize(1) poly16_t * ptr, poly16x4_t val, __constrange(0,3) int lane);
// VST1.16 {d0[0]}, [r0]

```

E.3.15 Loads and stores of an N-element structure

These intrinsics load or store an n -element structure. The array structures are defined similarly, for example the `int16x4x2_t` structure is defined as:

```

    struct int16x4x2_t
    {
        int16x4_t val[2];
    };

uint8x16x2_t vld2q_u8(__transfersize(32) uint8_t const * ptr);
// VLD2.8 {d0, d2}, [r0]
uint16x8x2_t vld2q_u16(__transfersize(16) uint16_t const * ptr);
// VLD2.16 {d0, d2}, [r0]
uint32x4x2_t vld2q_u32(__transfersize(8) uint32_t const * ptr);
// VLD2.32 {d0, d2}, [r0]
int8x16x2_t vld2q_s8(__transfersize(32) int8_t const * ptr);
// VLD2.8 {d0, d2}, [r0]
int16x8x2_t vld2q_s16(__transfersize(16) int16_t const * ptr);
// VLD2.16 {d0, d2}, [r0]
int32x4x2_t vld2q_s32(__transfersize(8) int32_t const * ptr);
// VLD2.32 {d0, d2}, [r0]
float32x4x2_t vld2q_f32(__transfersize(8) float32_t const * ptr);
// VLD2.32 {d0, d2}, [r0]
poly8x16x2_t vld2q_p8(__transfersize(32) poly8_t const * ptr);
// VLD2.8 {d0, d2}, [r0]
poly16x8x2_t vld2q_p16(__transfersize(16) poly16_t const * ptr);
// VLD2.16 {d0, d2}, [r0]
uint8x8x2_t vld2_u8(__transfersize(16) uint8_t const * ptr);
// VLD2.8 {d0, d1}, [r0]
uint16x4x2_t vld2_u16(__transfersize(8) uint16_t const * ptr);
// VLD2.16 {d0, d1}, [r0]
uint32x2x2_t vld2_u32(__transfersize(4) uint32_t const * ptr);
// VLD2.32 {d0, d1}, [r0]
uint64x1x2_t vld2_u64(__transfersize(2) uint64_t const * ptr);
// VLD1.64 {d0, d1}, [r0]
int8x8x2_t vld2_s8(__transfersize(16) int8_t const * ptr);
// VLD2.8 {d0, d1}, [r0]
int16x4x2_t vld2_s16(__transfersize(8) int16_t const * ptr);
// VLD2.16 {d0, d1}, [r0]
int32x2x2_t vld2_s32(__transfersize(4) int32_t const * ptr);
// VLD2.32 {d0, d1}, [r0]
int64x1x2_t vld2_s64(__transfersize(2) int64_t const * ptr);
// VLD1.64 {d0, d1}, [r0]
float32x2x2_t vld2_f32(__transfersize(4) float32_t const * ptr);
// VLD2.32 {d0, d1}, [r0]
poly8x8x2_t vld2_p8(__transfersize(16) poly8_t const * ptr);
// VLD2.8 {d0, d1}, [r0]
poly16x4x2_t vld2_p16(__transfersize(8) poly16_t const * ptr);
// VLD2.16 {d0, d1}, [r0]
uint8x16x3_t vld3q_u8(__transfersize(48) uint8_t const * ptr);
// VLD3.8 {d0, d2, d4}, [r0]
uint16x8x3_t vld3q_u16(__transfersize(24) uint16_t const * ptr);
// VLD3.16 {d0, d2, d4}, [r0]
uint32x4x3_t vld3q_u32(__transfersize(12) uint32_t const * ptr);

```

```

// VLD3.32 {d0, d2, d4}, [r0]
int8x16x3_t vld3q_s8(__transfersize(48) int8_t const * ptr);
// VLD3.8 {d0, d2, d4}, [r0]
int16x8x3_t vld3q_s16(__transfersize(24) int16_t const * ptr);
// VLD3.16 {d0, d2, d4}, [r0]
int32x4x3_t vld3q_s32(__transfersize(12) int32_t const * ptr);
// VLD3.32 {d0, d2, d4}, [r0]
float32x4x3_t vld3q_f32(__transfersize(12) float32_t const * ptr);
// VLD3.32 {d0, d2, d4}, [r0]
poly8x16x3_t vld3q_p8(__transfersize(48) poly8_t const * ptr);
// VLD3.8 {d0, d2, d4}, [r0]
poly16x8x3_t vld3q_p16(__transfersize(24) poly16_t const * ptr);
// VLD3.16 {d0, d2, d4}, [r0]
uint8x8x3_t vld3_u8(__transfersize(24) uint8_t const * ptr);
// VLD3.8 {d0, d1, d2}, [r0]
uint16x4x3_t vld3_u16(__transfersize(12) uint16_t const * ptr);
// VLD3.16 {d0, d1, d2}, [r0]
uint32x2x3_t vld3_u32(__transfersize(6) uint32_t const * ptr);
// VLD3.32 {d0, d1, d2}, [r0]
uint64x1x3_t vld3_u64(__transfersize(3) uint64_t const * ptr);
// VLD1.64 {d0, d1, d2}, [r0]
int8x8x3_t vld3_s8(__transfersize(24) int8_t const * ptr);
// VLD3.8 {d0, d1, d2}, [r0]
int16x4x3_t vld3_s16(__transfersize(12) int16_t const * ptr);
// VLD3.16 {d0, d1, d2}, [r0]
int32x2x3_t vld3_s32(__transfersize(6) int32_t const * ptr);
// VLD3.32 {d0, d1, d2}, [r0]
int64x1x3_t vld3_s64(__transfersize(3) int64_t const * ptr);
// VLD1.64 {d0, d1, d2}, [r0]
float32x2x3_t vld3_f32(__transfersize(6) float32_t const * ptr);
// VLD3.32 {d0, d1, d2}, [r0]
poly8x8x3_t vld3_p8(__transfersize(24) poly8_t const * ptr);
// VLD3.8 {d0, d1, d2}, [r0]
poly16x4x3_t vld3_p16(__transfersize(12) poly16_t const * ptr);
// VLD3.16 {d0, d1, d2}, [r0]
uint8x16x4_t vld4q_u8(__transfersize(64) uint8_t const * ptr);
// VLD4.8 {d0, d2, d4, d6}, [r0]
uint16x8x4_t vld4q_u16(__transfersize(32) uint16_t const * ptr);
// VLD4.16 {d0, d2, d4, d6}, [r0]
uint32x4x4_t vld4q_u32(__transfersize(16) uint32_t const * ptr);
// VLD4.32 {d0, d2, d4, d6}, [r0]
int8x16x4_t vld4q_s8(__transfersize(64) int8_t const * ptr);
// VLD4.8 {d0, d2, d4, d6}, [r0]
int16x8x4_t vld4q_s16(__transfersize(32) int16_t const * ptr);
// VLD4.16 {d0, d2, d4, d6}, [r0]
int32x4x4_t vld4q_s32(__transfersize(16) int32_t const * ptr);
// VLD4.32 {d0, d2, d4, d6}, [r0]
float32x4x4_t vld4q_f32(__transfersize(16) float32_t const * ptr);
// VLD4.32 {d0, d2, d4, d6}, [r0]
poly8x16x4_t vld4q_p8(__transfersize(64) poly8_t const * ptr);

```

```

// VLD4.8 {d0, d2, d4, d6}, [r0]
poly16x8x4_t vld4q_p16(__transfersize(32) poly16_t const * ptr);
// VLD4.16 {d0, d2, d4, d6}, [r0]
uint8x8x4_t vld4_u8(__transfersize(32) uint8_t const * ptr);
// VLD4.8 {d0, d1, d2, d3}, [r0]
uint16x4x4_t vld4_u16(__transfersize(16) uint16_t const * ptr);
// VLD4.16 {d0, d1, d2, d3}, [r0]
uint32x2x4_t vld4_u32(__transfersize(8) uint32_t const * ptr);
// VLD4.32 {d0, d1, d2, d3}, [r0]
uint64x1x4_t vld4_u64(__transfersize(4) uint64_t const * ptr);
// VLD1.64 {d0, d1, d2, d3}, [r0]
int8x8x4_t vld4_s8(__transfersize(32) int8_t const * ptr);
// VLD4.8 {d0, d1, d2, d3}, [r0]
int16x4x4_t vld4_s16(__transfersize(16) int16_t const * ptr);
// VLD4.16 {d0, d1, d2, d3}, [r0]
int32x2x4_t vld4_s32(__transfersize(8) int32_t const * ptr);
// VLD4.32 {d0, d1, d2, d3}, [r0]
int64x1x4_t vld4_s64(__transfersize(4) int64_t const * ptr);
// VLD1.64 {d0, d1, d2, d3}, [r0]
float32x2x4_t vld4_f32(__transfersize(8) float32_t const * ptr);
// VLD4.32 {d0, d1, d2, d3}, [r0]
poly8x8x4_t vld4_p8(__transfersize(32) poly8_t const * ptr);
// VLD4.8 {d0, d1, d2, d3}, [r0]
poly16x4x4_t vld4_p16(__transfersize(16) poly16_t const * ptr);
// VLD4.16 {d0, d1, d2, d3}, [r0]
uint8x8x2_t vld2_dup_u8(__transfersize(2) uint8_t const * ptr);
// VLD2.8 {d0[], d1[]}, [r0]
uint16x4x2_t vld2_dup_u16(__transfersize(2) uint16_t const * ptr);
// VLD2.16 {d0[], d1[]}, [r0]
uint32x2x2_t vld2_dup_u32(__transfersize(2) uint32_t const * ptr);
// VLD2.32 {d0[], d1[]}, [r0]
uint64x1x2_t vld2_dup_u64(__transfersize(2) uint64_t const * ptr);
// VLD1.64 {d0, d1}, [r0]
int8x8x2_t vld2_dup_s8(__transfersize(2) int8_t const * ptr);
// VLD2.8 {d0[], d1[]}, [r0]
int16x4x2_t vld2_dup_s16(__transfersize(2) int16_t const * ptr);
// VLD2.16 {d0[], d1[]}, [r0]
int32x2x2_t vld2_dup_s32(__transfersize(2) int32_t const * ptr);
// VLD2.32 {d0[], d1[]}, [r0]
int64x1x2_t vld2_dup_s64(__transfersize(2) int64_t const * ptr);
// VLD1.64 {d0, d1}, [r0]
float32x2x2_t vld2_dup_f32(__transfersize(2) float32_t const * ptr);
// VLD2.32 {d0[], d1[]}, [r0]
poly8x8x2_t vld2_dup_p8(__transfersize(2) poly8_t const * ptr);
// VLD2.8 {d0[], d1[]}, [r0]
poly16x4x2_t vld2_dup_p16(__transfersize(2) poly16_t const * ptr);
// VLD2.16 {d0[], d1[]}, [r0]
uint8x8x3_t vld3_dup_u8(__transfersize(3) uint8_t const * ptr);
// VLD3.8 {d0[], d1[], d2[]}, [r0]
uint16x4x3_t vld3_dup_u16(__transfersize(3) uint16_t const * ptr);

```

```

// VLD3.16 {d0[], d1[], d2[]}, [r0]
uint32x2x3_t vld3_dup_u32(__transfersize(3) uint32_t const * ptr);
// VLD3.32 {d0[], d1[], d2[]}, [r0]
uint64x1x3_t vld3_dup_u64(__transfersize(3) uint64_t const * ptr);
// VLD1.64 {d0, d1, d2}, [r0]
int8x8x3_t vld3_dup_s8(__transfersize(3) int8_t const * ptr);
// VLD3.8 {d0[], d1[], d2[]}, [r0]
int16x4x3_t vld3_dup_s16(__transfersize(3) int16_t const * ptr);
// VLD3.16 {d0[], d1[], d2[]}, [r0]
int32x2x3_t vld3_dup_s32(__transfersize(3) int32_t const * ptr);
// VLD3.32 {d0[], d1[], d2[]}, [r0]
int64x1x3_t vld3_dup_s64(__transfersize(3) int64_t const * ptr);
// VLD1.64 {d0, d1, d2}, [r0]
float32x2x3_t vld3_dup_f32(__transfersize(3) float32_t const * ptr);
// VLD3.32 {d0[], d1[], d2[]}, [r0]
poly8x8x3_t vld3_dup_p8(__transfersize(3) poly8_t const * ptr);
// VLD3.8 {d0[], d1[], d2[]}, [r0]
poly16x4x3_t vld3_dup_p16(__transfersize(3) poly16_t const * ptr);
// VLD3.16 {d0[], d1[], d2[]}, [r0]
uint8x8x4_t vld4_dup_u8(__transfersize(4) uint8_t const * ptr);
// VLD4.8 {d0[], d1[], d2[], d3[]}, [r0]
uint16x4x4_t vld4_dup_u16(__transfersize(4) uint16_t const * ptr);
// VLD4.16 {d0[], d1[], d2[], d3[]}, [r0]
uint32x2x4_t vld4_dup_u32(__transfersize(4) uint32_t const * ptr);
// VLD4.32 {d0[], d1[], d2[], d3[]}, [r0]
uint64x1x4_t vld4_dup_u64(__transfersize(4) uint64_t const * ptr);
// VLD1.64 {d0, d1, d2, d3}, [r0]
int8x8x4_t vld4_dup_s8(__transfersize(4) int8_t const * ptr);
// VLD4.8 {d0[], d1[], d2[], d3[]}, [r0]
int16x4x4_t vld4_dup_s16(__transfersize(4) int16_t const * ptr);
// VLD4.16 {d0[], d1[], d2[], d3[]}, [r0]
int32x2x4_t vld4_dup_s32(__transfersize(4) int32_t const * ptr);
// VLD4.32 {d0[], d1[], d2[], d3[]}, [r0]
int64x1x4_t vld4_dup_s64(__transfersize(4) int64_t const * ptr);
// VLD1.64 {d0, d1, d2, d3}, [r0]
float32x2x4_t vld4_dup_f32(__transfersize(4) float32_t const * ptr);
// VLD4.32 {d0[], d1[], d2[], d3[]}, [r0]
poly8x8x4_t vld4_dup_p8(__transfersize(4) poly8_t const * ptr);
// VLD4.8 {d0[], d1[], d2[], d3[]}, [r0]
poly16x4x4_t vld4_dup_p16(__transfersize(4) poly16_t const * ptr);
// VLD4.16 {d0[], d1[], d2[], d3[]}, [r0]
uint16x8x2_t vld2q_lane_u16(__transfersize(2) uint16_t const * ptr, uint16x8x2_t src,
__constrange(0,7) int lane);
// VLD2.16 {d0[0], d2[0]}, [r0]
uint32x4x2_t vld2q_lane_u32(__transfersize(2) uint32_t const * ptr, uint32x4x2_t src,
__constrange(0,3) int lane);
// VLD2.32 {d0[0], d2[0]}, [r0]
int16x8x2_t vld2q_lane_s16(__transfersize(2) int16_t const * ptr, int16x8x2_t src, __constrange(0,7)
int lane);
// VLD2.16 {d0[0], d2[0]}, [r0]

```

```

int32x4x2_t vld2q_lane_s32(__transfersize(2) int32_t const * ptr, int32x4x2_t src, __constrange(0,3)
int lane);
// VLD2.32 {d0[0], d2[0]}, [r0]
float32x4x2_t vld2q_lane_f32(__transfersize(2) float32_t const * ptr, float32x4x2_t src,
__constrange(0,3) int lane);
// VLD2.32 {d0[0], d2[0]}, [r0]
poly16x8x2_t vld2q_lane_p16(__transfersize(2) poly16_t const * ptr, poly16x8x2_t src,
__constrange(0,7) int lane);
// VLD2.16 {d0[0], d2[0]}, [r0]
uint8x8x2_t vld2_lane_u8(__transfersize(2) uint8_t const * ptr, uint8x8x2_t src, __constrange(0,7) int
lane);
// VLD2.8 {d0[0], d1[0]}, [r0]
uint16x4x2_t vld2_lane_u16(__transfersize(2) uint16_t const * ptr, uint16x4x2_t src, __constrange(0,3)
int lane);
// VLD2.16 {d0[0], d1[0]}, [r0]
uint32x2x2_t vld2_lane_u32(__transfersize(2) uint32_t const * ptr, uint32x2x2_t src, __constrange(0,1)
int lane);
// VLD2.32 {d0[0], d1[0]}, [r0]
int8x8x2_t vld2_lane_s8(__transfersize(2) int8_t const * ptr, int8x8x2_t src, __constrange(0,7) int
lane);
// VLD2.8 {d0[0], d1[0]}, [r0]
int16x4x2_t vld2_lane_s16(__transfersize(2) int16_t const * ptr, int16x4x2_t src, __constrange(0,3) int
lane);
// VLD2.16 {d0[0], d1[0]}, [r0]
int32x2x2_t vld2_lane_s32(__transfersize(2) int32_t const * ptr, int32x2x2_t src, __constrange(0,1) int
lane);
// VLD2.32 {d0[0], d1[0]}, [r0]
float32x2x2_t vld2_lane_f32(__transfersize(2) float32_t const * ptr, float32x2x2_t src,
__constrange(0,1) int lane);
// VLD2.32 {d0[0], d1[0]}, [r0]
poly8x8x2_t vld2_lane_p8(__transfersize(2) poly8_t const * ptr, poly8x8x2_t src, __constrange(0,7) int
lane);
// VLD2.8 {d0[0], d1[0]}, [r0]
poly16x4x2_t vld2_lane_p16(__transfersize(2) poly16_t const * ptr, poly16x4x2_t src, __constrange(0,3)
int lane);
// VLD2.16 {d0[0], d1[0]}, [r0]
uint16x8x3_t vld3q_lane_u16(__transfersize(3) uint16_t const * ptr, uint16x8x3_t src,
__constrange(0,7) int lane);
// VLD3.16 {d0[0], d2[0], d4[0]}, [r0]
uint32x4x3_t vld3q_lane_u32(__transfersize(3) uint32_t const * ptr, uint32x4x3_t src,
__constrange(0,3) int lane);
// VLD3.32 {d0[0], d2[0], d4[0]}, [r0]
int16x8x3_t vld3q_lane_s16(__transfersize(3) int16_t const * ptr, int16x8x3_t src, __constrange(0,7)
int lane);
// VLD3.16 {d0[0], d2[0], d4[0]}, [r0]
int32x4x3_t vld3q_lane_s32(__transfersize(3) int32_t const * ptr, int32x4x3_t src, __constrange(0,3)
int lane);
// VLD3.32 {d0[0], d2[0], d4[0]}, [r0]
float32x4x3_t vld3q_lane_f32(__transfersize(3) float32_t const * ptr, float32x4x3_t src,
__constrange(0,3) int lane);

```

```

// VLD3.32 {d0[0], d2[0], d4[0]}, [r0]
poly16x8x3_t vld3q_lane_p16(__transfersize(3) poly16_t const * ptr, poly16x8x3_t src,
__constrange(0,7) int lane);

// VLD3.16 {d0[0], d2[0], d4[0]}, [r0]
uint8x8x3_t vld3_lane_u8(__transfersize(3) uint8_t const * ptr, uint8x8x3_t src, __constrange(0,7) int
lane);

// VLD3.8 {d0[0], d1[0], d2[0]}, [r0]
uint16x4x3_t vld3_lane_u16(__transfersize(3) uint16_t const * ptr, uint16x4x3_t src, __constrange(0,3)
int lane);

// VLD3.16 {d0[0], d1[0], d2[0]}, [r0]
uint32x2x3_t vld3_lane_u32(__transfersize(3) uint32_t const * ptr, uint32x2x3_t src, __constrange(0,1)
int lane);

// VLD3.32 {d0[0], d1[0], d2[0]}, [r0]
int8x8x3_t vld3_lane_s8(__transfersize(3) int8_t const * ptr, int8x8x3_t src, __constrange(0,7) int
lane);

// VLD3.8 {d0[0], d1[0], d2[0]}, [r0]
int16x4x3_t vld3_lane_s16(__transfersize(3) int16_t const * ptr, int16x4x3_t src, __constrange(0,3) int
lane);

// VLD3.16 {d0[0], d1[0], d2[0]}, [r0]
int32x2x3_t vld3_lane_s32(__transfersize(3) int32_t const * ptr, int32x2x3_t src, __constrange(0,1) int
lane);

// VLD3.32 {d0[0], d1[0], d2[0]}, [r0]
float32x2x3_t vld3_lane_f32(__transfersize(3) float32_t const * ptr, float32x2x3_t src,
__constrange(0,1) int lane);

// VLD3.32 {d0[0], d1[0], d2[0]}, [r0]
poly8x8x3_t vld3_lane_p8(__transfersize(3) poly8_t const * ptr, poly8x8x3_t src, __constrange(0,7) int
lane);

// VLD3.8 {d0[0], d1[0], d2[0]}, [r0]
poly16x4x3_t vld3_lane_p16(__transfersize(3) poly16_t const * ptr, poly16x4x3_t src, __constrange(0,3)
int lane);

// VLD3.16 {d0[0], d1[0], d2[0]}, [r0]
uint16x8x4_t vld4q_lane_u16(__transfersize(4) uint16_t const * ptr, uint16x8x4_t src,
__constrange(0,7) int lane);

// VLD4.16 {d0[0], d2[0], d4[0], d6[0]}, [r0]
uint32x4x4_t vld4q_lane_u32(__transfersize(4) uint32_t const * ptr, uint32x4x4_t src,
__constrange(0,3) int lane);

// VLD4.32 {d0[0], d2[0], d4[0], d6[0]}, [r0]
int16x8x4_t vld4q_lane_s16(__transfersize(4) int16_t const * ptr, int16x8x4_t src, __constrange(0,7)
int lane);

// VLD4.16 {d0[0], d2[0], d4[0], d6[0]}, [r0]
int32x4x4_t vld4q_lane_s32(__transfersize(4) int32_t const * ptr, int32x4x4_t src, __constrange(0,3)
int lane);

// VLD4.32 {d0[0], d2[0], d4[0], d6[0]}, [r0]
float32x4x4_t vld4q_lane_f32(__transfersize(4) float32_t const * ptr, float32x4x4_t src,
__constrange(0,3) int lane);

// VLD4.32 {d0[0], d2[0], d4[0], d6[0]}, [r0]
poly16x8x4_t vld4q_lane_p16(__transfersize(4) poly16_t const * ptr, poly16x8x4_t src,
__constrange(0,7) int lane);

// VLD4.16 {d0[0], d2[0], d4[0], d6[0]}, [r0]
uint8x8x4_t vld4_lane_u8(__transfersize(4) uint8_t const * ptr, uint8x8x4_t src, __constrange(0,7) int

```

```

lane);
                                // VLD4.8 {d0[0], d1[0], d2[0], d3[0]}, [r0]
uint16x4x4_t vld4_lane_u16(__transfersize(4) uint16_t const * ptr, uint16x4x4_t src, __constrange(0,3)
int lane);
                                // VLD4.16 {d0[0], d1[0], d2[0], d3[0]}, [r0]
uint32x2x4_t vld4_lane_u32(__transfersize(4) uint32_t const * ptr, uint32x2x4_t src, __constrange(0,1)
int lane);
                                // VLD4.32 {d0[0], d1[0], d2[0], d3[0]}, [r0]
int8x8x4_t vld4_lane_s8(__transfersize(4) int8_t const * ptr, int8x8x4_t src, __constrange(0,7) int
lane);
                                // VLD4.8 {d0[0], d1[0], d2[0], d3[0]}, [r0]
int16x4x4_t vld4_lane_s16(__transfersize(4) int16_t const * ptr, int16x4x4_t src, __constrange(0,3) int
lane);
                                // VLD4.16 {d0[0], d1[0], d2[0], d3[0]}, [r0]
int32x2x4_t vld4_lane_s32(__transfersize(4) int32_t const * ptr, int32x2x4_t src, __constrange(0,1) int
lane);
                                // VLD4.32 {d0[0], d1[0], d2[0], d3[0]}, [r0]
float32x2x4_t vld4_lane_f32(__transfersize(4) float32_t const * ptr, float32x2x4_t src,
__constrange(0,1) int lane);
                                // VLD4.32 {d0[0], d1[0], d2[0], d3[0]}, [r0]
poly8x8x4_t vld4_lane_p8(__transfersize(4) poly8_t const * ptr, poly8x8x4_t src, __constrange(0,7) int
lane);
                                // VLD4.8 {d0[0], d1[0], d2[0], d3[0]}, [r0]
poly16x4x4_t vld4_lane_p16(__transfersize(4) poly16_t const * ptr, poly16x4x4_t src, __constrange(0,3)
int lane);
                                // VLD4.16 {d0[0], d1[0], d2[0], d3[0]}, [r0]
void vst2q_u8(__transfersize(32) uint8_t * ptr, uint8x16x2_t val);
                                // VST2.8 {d0, d2}, [r0]
void vst2q_u16(__transfersize(16) uint16_t * ptr, uint16x8x2_t val);
                                // VST2.16 {d0, d2}, [r0]
void vst2q_u32(__transfersize(8) uint32_t * ptr, uint32x4x2_t val);
                                // VST2.32 {d0, d2}, [r0]
void vst2q_s8(__transfersize(32) int8_t * ptr, int8x16x2_t val);
                                // VST2.8 {d0, d2}, [r0]
void vst2q_s16(__transfersize(16) int16_t * ptr, int16x8x2_t val);
                                // VST2.16 {d0, d2}, [r0]
void vst2q_s32(__transfersize(8) int32_t * ptr, int32x4x2_t val);
                                // VST2.32 {d0, d2}, [r0]
void vst2q_f32(__transfersize(8) float32_t * ptr, float32x4x2_t val);
                                // VST2.32 {d0, d2}, [r0]
void vst2q_p8(__transfersize(32) poly8_t * ptr, poly8x16x2_t val);
                                // VST2.8 {d0, d2}, [r0]
void vst2q_p16(__transfersize(16) poly16_t * ptr, poly16x8x2_t val);
                                // VST2.16 {d0, d2}, [r0]
void vst2_u8(__transfersize(16) uint8_t * ptr, uint8x8x2_t val);
                                // VST2.8 {d0, d1}, [r0]
void vst2_u16(__transfersize(8) uint16_t * ptr, uint16x4x2_t val);
                                // VST2.16 {d0, d1}, [r0]
void vst2_u32(__transfersize(4) uint32_t * ptr, uint32x2x2_t val);
                                // VST2.32 {d0, d1}, [r0]

```

```

void vst2_u64(__transfersize(2) uint64_t * ptr, uint64x1x2_t val);
// VST1.64 {d0, d1}, [r0]
void vst2_s8(__transfersize(16) int8_t * ptr, int8x8x2_t val);
// VST2.8 {d0, d1}, [r0]
void vst2_s16(__transfersize(8) int16_t * ptr, int16x4x2_t val);
// VST2.16 {d0, d1}, [r0]
void vst2_s32(__transfersize(4) int32_t * ptr, int32x2x2_t val);
// VST2.32 {d0, d1}, [r0]
void vst2_s64(__transfersize(2) int64_t * ptr, int64x1x2_t val);
// VST1.64 {d0, d1}, [r0]
void vst2_f32(__transfersize(4) float32_t * ptr, float32x2x2_t val);
// VST2.32 {d0, d1}, [r0]
void vst2_p8(__transfersize(16) poly8_t * ptr, poly8x8x2_t val);
// VST2.8 {d0, d1}, [r0]
void vst2_p16(__transfersize(8) poly16_t * ptr, poly16x4x2_t val);
// VST2.16 {d0, d1}, [r0]
void vst3q_u8(__transfersize(48) uint8_t * ptr, uint8x16x3_t val);
// VST3.8 {d0, d2, d4}, [r0]
void vst3q_u16(__transfersize(24) uint16_t * ptr, uint16x8x3_t val);
// VST3.16 {d0, d2, d4}, [r0]
void vst3q_u32(__transfersize(12) uint32_t * ptr, uint32x4x3_t val);
// VST3.32 {d0, d2, d4}, [r0]
void vst3q_s8(__transfersize(48) int8_t * ptr, int8x16x3_t val);
// VST3.8 {d0, d2, d4}, [r0]
void vst3q_s16(__transfersize(24) int16_t * ptr, int16x8x3_t val);
// VST3.16 {d0, d2, d4}, [r0]
void vst3q_s32(__transfersize(12) int32_t * ptr, int32x4x3_t val);
// VST3.32 {d0, d2, d4}, [r0]
void vst3q_f32(__transfersize(12) float32_t * ptr, float32x4x3_t val);
// VST3.32 {d0, d2, d4}, [r0]
void vst3q_p8(__transfersize(48) poly8_t * ptr, poly8x16x3_t val);
// VST3.8 {d0, d2, d4}, [r0]
void vst3q_p16(__transfersize(24) poly16_t * ptr, poly16x8x3_t val);
// VST3.16 {d0, d2, d4}, [r0]
void vst3_u8(__transfersize(24) uint8_t * ptr, uint8x8x3_t val);
// VST3.8 {d0, d1, d2}, [r0]
void vst3_u16(__transfersize(12) uint16_t * ptr, uint16x4x3_t val);
// VST3.16 {d0, d1, d2}, [r0]
void vst3_u32(__transfersize(6) uint32_t * ptr, uint32x2x3_t val);
// VST3.32 {d0, d1, d2}, [r0]
void vst3_u64(__transfersize(3) uint64_t * ptr, uint64x1x3_t val);
// VST1.64 {d0, d1, d2}, [r0]
void vst3_s8(__transfersize(24) int8_t * ptr, int8x8x3_t val);
// VST3.8 {d0, d1, d2}, [r0]
void vst3_s16(__transfersize(12) int16_t * ptr, int16x4x3_t val);
// VST3.16 {d0, d1, d2}, [r0]
void vst3_s32(__transfersize(6) int32_t * ptr, int32x2x3_t val);
// VST3.32 {d0, d1, d2}, [r0]
void vst3_s64(__transfersize(3) int64_t * ptr, int64x1x3_t val);
// VST1.64 {d0, d1, d2}, [r0]

```

```

void vst3_f32(__transfersize(6) float32_t * ptr, float32x2x3_t val);
// VST3.32 {d0, d1, d2}, [r0]
void vst3_p8(__transfersize(24) poly8_t * ptr, poly8x8x3_t val);
// VST3.8 {d0, d1, d2}, [r0]
void vst3_p16(__transfersize(12) poly16_t * ptr, poly16x4x3_t val);
// VST3.16 {d0, d1, d2}, [r0]
void vst4q_u8(__transfersize(64) uint8_t * ptr, uint8x16x4_t val);
// VST4.8 {d0, d2, d4, d6}, [r0]
void vst4q_u16(__transfersize(32) uint16_t * ptr, uint16x8x4_t val);
// VST4.16 {d0, d2, d4, d6}, [r0]
void vst4q_u32(__transfersize(16) uint32_t * ptr, uint32x4x4_t val);
// VST4.32 {d0, d2, d4, d6}, [r0]
void vst4q_s8(__transfersize(64) int8_t * ptr, int8x16x4_t val);
// VST4.8 {d0, d2, d4, d6}, [r0]
void vst4q_s16(__transfersize(32) int16_t * ptr, int16x8x4_t val);
// VST4.16 {d0, d2, d4, d6}, [r0]
void vst4q_s32(__transfersize(16) int32_t * ptr, int32x4x4_t val);
// VST4.32 {d0, d2, d4, d6}, [r0]
void vst4q_f32(__transfersize(16) float32_t * ptr, float32x4x4_t val);
// VST4.32 {d0, d2, d4, d6}, [r0]
void vst4q_p8(__transfersize(64) poly8_t * ptr, poly8x16x4_t val);
// VST4.8 {d0, d2, d4, d6}, [r0]
void vst4q_p16(__transfersize(32) poly16_t * ptr, poly16x8x4_t val);
// VST4.16 {d0, d2, d4, d6}, [r0]
void vst4_u8(__transfersize(32) uint8_t * ptr, uint8x8x4_t val);
// VST4.8 {d0, d1, d2, d3}, [r0]
void vst4_u16(__transfersize(16) uint16_t * ptr, uint16x4x4_t val);
// VST4.16 {d0, d1, d2, d3}, [r0]
void vst4_u32(__transfersize(8) uint32_t * ptr, uint32x2x4_t val);
// VST4.32 {d0, d1, d2, d3}, [r0]
void vst4_u64(__transfersize(4) uint64_t * ptr, uint64x1x4_t val);
// VST1.64 {d0, d1, d2, d3}, [r0]
void vst4_s8(__transfersize(32) int8_t * ptr, int8x8x4_t val);
// VST4.8 {d0, d1, d2, d3}, [r0]
void vst4_s16(__transfersize(16) int16_t * ptr, int16x4x4_t val);
// VST4.16 {d0, d1, d2, d3}, [r0]
void vst4_s32(__transfersize(8) int32_t * ptr, int32x2x4_t val);
// VST4.32 {d0, d1, d2, d3}, [r0]
void vst4_s64(__transfersize(4) int64_t * ptr, int64x1x4_t val);
// VST1.64 {d0, d1, d2, d3}, [r0]
void vst4_f32(__transfersize(8) float32_t * ptr, float32x2x4_t val);
// VST4.32 {d0, d1, d2, d3}, [r0]
void vst4_p8(__transfersize(32) poly8_t * ptr, poly8x8x4_t val);
// VST4.8 {d0, d1, d2, d3}, [r0]
void vst4_p16(__transfersize(16) poly16_t * ptr, poly16x4x4_t val);
// VST4.16 {d0, d1, d2, d3}, [r0]
void vst2q_lane_u16(__transfersize(2) uint16_t * ptr, uint16x8x2_t val, __constrange(0,7) int lane);
// VST2.16 {d0[0], d2[0]}, [r0]
void vst2q_lane_u32(__transfersize(2) uint32_t * ptr, uint32x4x2_t val, __constrange(0,3) int lane);
// VST2.32 {d0[0], d2[0]}, [r0]

```

```

void vst2q_lane_s16(__transfersize(2) int16_t * ptr, int16x8x2_t val, __constrange(0,7) int lane);
// VST2.16 {d0[0], d2[0]}, [r0]
void vst2q_lane_s32(__transfersize(2) int32_t * ptr, int32x4x2_t val, __constrange(0,3) int lane);
// VST2.32 {d0[0], d2[0]}, [r0]
void vst2q_lane_f32(__transfersize(2) float32_t * ptr, float32x4x2_t val, __constrange(0,3) int lane);
// VST2.32 {d0[0], d2[0]}, [r0]
void vst2q_lane_p16(__transfersize(2) poly16_t * ptr, poly16x8x2_t val, __constrange(0,7) int lane);
// VST2.16 {d0[0], d2[0]}, [r0]
void vst2_lane_u8(__transfersize(2) uint8_t * ptr, uint8x8x2_t val, __constrange(0,7) int lane);
// VST2.8 {d0[0], d1[0]}, [r0]
void vst2_lane_u16(__transfersize(2) uint16_t * ptr, uint16x4x2_t val, __constrange(0,3) int lane);
// VST2.16 {d0[0], d1[0]}, [r0]
void vst2_lane_u32(__transfersize(2) uint32_t * ptr, uint32x2x2_t val, __constrange(0,1) int lane);
// VST2.32 {d0[0], d1[0]}, [r0]
void vst2_lane_s8(__transfersize(2) int8_t * ptr, int8x8x2_t val, __constrange(0,7) int lane);
// VST2.8 {d0[0], d1[0]}, [r0]
void vst2_lane_s16(__transfersize(2) int16_t * ptr, int16x4x2_t val, __constrange(0,3) int lane);
// VST2.16 {d0[0], d1[0]}, [r0]
void vst2_lane_s32(__transfersize(2) int32_t * ptr, int32x2x2_t val, __constrange(0,1) int lane);
// VST2.32 {d0[0], d1[0]}, [r0]
void vst2_lane_f32(__transfersize(2) float32_t * ptr, float32x2x2_t val, __constrange(0,1) int lane);
// VST2.32 {d0[0], d1[0]}, [r0]
void vst2_lane_p8(__transfersize(2) poly8_t * ptr, poly8x8x2_t val, __constrange(0,7) int lane);
// VST2.8 {d0[0], d1[0]}, [r0]
void vst2_lane_p16(__transfersize(2) poly16_t * ptr, poly16x4x2_t val, __constrange(0,3) int lane);
// VST2.16 {d0[0], d1[0]}, [r0]
void vst3q_lane_u16(__transfersize(3) uint16_t * ptr, uint16x8x3_t val, __constrange(0,7) int lane);
// VST3.16 {d0[0], d2[0], d4[0]}, [r0]
void vst3q_lane_u32(__transfersize(3) uint32_t * ptr, uint32x4x3_t val, __constrange(0,3) int lane);
// VST3.32 {d0[0], d2[0], d4[0]}, [r0]
void vst3q_lane_s16(__transfersize(3) int16_t * ptr, int16x8x3_t val, __constrange(0,7) int lane);
// VST3.16 {d0[0], d2[0], d4[0]}, [r0]
void vst3q_lane_s32(__transfersize(3) int32_t * ptr, int32x4x3_t val, __constrange(0,3) int lane);
// VST3.32 {d0[0], d2[0], d4[0]}, [r0]
void vst3q_lane_f32(__transfersize(3) float32_t * ptr, float32x4x3_t val, __constrange(0,3) int lane);
// VST3.32 {d0[0], d2[0], d4[0]}, [r0]
void vst3q_lane_p16(__transfersize(3) poly16_t * ptr, poly16x8x3_t val, __constrange(0,7) int lane);
// VST3.16 {d0[0], d2[0], d4[0]}, [r0]
void vst3_lane_u8(__transfersize(3) uint8_t * ptr, uint8x8x3_t val, __constrange(0,7) int lane);
// VST3.8 {d0[0], d1[0], d2[0]}, [r0]
void vst3_lane_u16(__transfersize(3) uint16_t * ptr, uint16x4x3_t val, __constrange(0,3) int lane);
// VST3.16 {d0[0], d1[0], d2[0]}, [r0]
void vst3_lane_u32(__transfersize(3) uint32_t * ptr, uint32x2x3_t val, __constrange(0,1) int lane);
// VST3.32 {d0[0], d1[0], d2[0]}, [r0]
void vst3_lane_s8(__transfersize(3) int8_t * ptr, int8x8x3_t val, __constrange(0,7) int lane);
// VST3.8 {d0[0], d1[0], d2[0]}, [r0]
void vst3_lane_s16(__transfersize(3) int16_t * ptr, int16x4x3_t val, __constrange(0,3) int lane);
// VST3.16 {d0[0], d1[0], d2[0]}, [r0]
void vst3_lane_s32(__transfersize(3) int32_t * ptr, int32x2x3_t val, __constrange(0,1) int lane);
// VST3.32 {d0[0], d1[0], d2[0]}, [r0]

```

```

void vst3_lane_f32(__transfersize(3) float32_t * ptr, float32x2x3_t val, __constrange(0,1) int lane);
// VST3.32 {d0[0], d1[0], d2[0]}, [r0]
void vst3_lane_p8(__transfersize(3) poly8_t * ptr, poly8x8x3_t val, __constrange(0,7) int lane);
// VST3.8 {d0[0], d1[0], d2[0]}, [r0]
void vst3_lane_p16(__transfersize(3) poly16_t * ptr, poly16x4x3_t val, __constrange(0,3) int lane);
// VST3.16 {d0[0], d1[0], d2[0]}, [r0]
void vst4q_lane_u16(__transfersize(4) uint16_t * ptr, uint16x8x4_t val, __constrange(0,7) int lane);
// VST4.16 {d0[0], d2[0], d4[0], d6[0]}, [r0]
void vst4q_lane_u32(__transfersize(4) uint32_t * ptr, uint32x4x4_t val, __constrange(0,3) int lane);
// VST4.32 {d0[0], d2[0], d4[0], d6[0]}, [r0]
void vst4q_lane_s16(__transfersize(4) int16_t * ptr, int16x8x4_t val, __constrange(0,7) int lane);
// VST4.16 {d0[0], d2[0], d4[0], d6[0]}, [r0]
void vst4q_lane_s32(__transfersize(4) int32_t * ptr, int32x4x4_t val, __constrange(0,3) int lane);
// VST4.32 {d0[0], d2[0], d4[0], d6[0]}, [r0]
void vst4q_lane_f32(__transfersize(4) float32_t * ptr, float32x4x4_t val, __constrange(0,3) int lane);
// VST4.32 {d0[0], d2[0], d4[0], d6[0]}, [r0]
void vst4q_lane_p16(__transfersize(4) poly16_t * ptr, poly16x8x4_t val, __constrange(0,7) int lane);
// VST4.16 {d0[0], d2[0], d4[0], d6[0]}, [r0]
void vst4_lane_u8(__transfersize(4) uint8_t * ptr, uint8x8x4_t val, __constrange(0,7) int lane);
// VST4.8 {d0[0], d1[0], d2[0], d3[0]}, [r0]
void vst4_lane_u16(__transfersize(4) uint16_t * ptr, uint16x4x4_t val, __constrange(0,3) int lane);
// VST4.16 {d0[0], d1[0], d2[0], d3[0]}, [r0]
void vst4_lane_u32(__transfersize(4) uint32_t * ptr, uint32x2x4_t val, __constrange(0,1) int lane);
// VST4.32 {d0[0], d1[0], d2[0], d3[0]}, [r0]
void vst4_lane_s8(__transfersize(4) int8_t * ptr, int8x8x4_t val, __constrange(0,7) int lane);
// VST4.8 {d0[0], d1[0], d2[0], d3[0]}, [r0]
void vst4_lane_s16(__transfersize(4) int16_t * ptr, int16x4x4_t val, __constrange(0,3) int lane);
// VST4.16 {d0[0], d1[0], d2[0], d3[0]}, [r0]
void vst4_lane_s32(__transfersize(4) int32_t * ptr, int32x2x4_t val, __constrange(0,1) int lane);
// VST4.32 {d0[0], d1[0], d2[0], d3[0]}, [r0]
void vst4_lane_f32(__transfersize(4) float32_t * ptr, float32x2x4_t val, __constrange(0,1) int lane);
// VST4.32 {d0[0], d1[0], d2[0], d3[0]}, [r0]
void vst4_lane_p8(__transfersize(4) poly8_t * ptr, poly8x8x4_t val, __constrange(0,7) int lane);
// VST4.8 {d0[0], d1[0], d2[0], d3[0]}, [r0]
void vst4_lane_p16(__transfersize(4) poly16_t * ptr, poly16x4x4_t val, __constrange(0,3) int lane);
// VST4.16 {d0[0], d1[0], d2[0], d3[0]}, [r0]

```

E.3.16 Extract lanes from a vector

These intrinsics extract a single lane (element) from a vector.

```

uint8_t  vget_lane_u8(uint8x8_t vec, __constrange(0,7) int lane);    // VMOV.U8 r0, d0[0]
uint16_t vget_lane_u16(uint16x4_t vec, __constrange(0,3) int lane);  // VMOV.U16 r0, d0[0]
uint32_t vget_lane_u32(uint32x2_t vec, __constrange(0,1) int lane);  // VMOV.32 r0, d0[0]
int8_t   vget_lane_s8(int8x8_t vec, __constrange(0,7) int lane);    // VMOV.S8 r0, d0[0]
int16_t  vget_lane_s16(int16x4_t vec, __constrange(0,3) int lane);   // VMOV.S16 r0, d0[0]
int32_t  vget_lane_s32(int32x2_t vec, __constrange(0,1) int lane);   // VMOV.32 r0, d0[0]
poly8_t  vget_lane_p8(poly8x8_t vec, __constrange(0,7) int lane);    // VMOV.U8 r0, d0[0]
poly16_t vget_lane_p16(poly16x4_t vec, __constrange(0,3) int lane);  // VMOV.U16 r0, d0[0]

```

```

float32_t vget_lane_f32(float32x2_t vec, __constrange(0,1) int lane); // VMOV.32 r0, d0[0]
uint8_t vgetq_lane_u8(uint8x16_t vec, __constrange(0,15) int lane); // VMOV.U8 r0, d0[0]
uint16_t vgetq_lane_u16(uint16x8_t vec, __constrange(0,7) int lane); // VMOV.U16 r0, d0[0]
uint32_t vgetq_lane_u32(uint32x4_t vec, __constrange(0,3) int lane); // VMOV.32 r0, d0[0]
int8_t vgetq_lane_s8(int8x16_t vec, __constrange(0,15) int lane); // VMOV.S8 r0, d0[0]
int16_t vgetq_lane_s16(int16x8_t vec, __constrange(0,7) int lane); // VMOV.S16 r0, d0[0]
int32_t vgetq_lane_s32(int32x4_t vec, __constrange(0,3) int lane); // VMOV.32 r0, d0[0]
poly8_t vgetq_lane_p8(poly8x16_t vec, __constrange(0,15) int lane); // VMOV.U8 r0, d0[0]
poly16_t vgetq_lane_p16(poly16x8_t vec, __constrange(0,7) int lane); // VMOV.U16 r0, d0[0]
float32_t vgetq_lane_f32(float32x4_t vec, __constrange(0,3) int lane); // VMOV.32 r0, d0[0]
int64_t vget_lane_s64(int64x1_t vec, __constrange(0,0) int lane); // VMOV r0,r0,d0
uint64_t vget_lane_u64(uint64x1_t vec, __constrange(0,0) int lane); // VMOV r0,r0,d0
int64_t vgetq_lane_s64(int64x2_t vec, __constrange(0,1) int lane); // VMOV r0,r0,d0
uint64_t vgetq_lane_u64(uint64x2_t vec, __constrange(0,1) int lane); // VMOV r0,r0,d0

```

E.3.17 Set lanes within a vector

These intrinsics set a single lane (element) within a vector.

```

uint8x8_t vset_lane_u8(uint8_t value, uint8x8_t vec, __constrange(0,7) int lane);
// VMOV.8 d0[0],r0
uint16x4_t vset_lane_u16(uint16_t value, uint16x4_t vec, __constrange(0,3) int lane);
// VMOV.16 d0[0],r0
uint32x2_t vset_lane_u32(uint32_t value, uint32x2_t vec, __constrange(0,1) int lane);
// VMOV.32 d0[0],r0
int8x8_t vset_lane_s8(int8_t value, int8x8_t vec, __constrange(0,7) int lane);
// VMOV.8 d0[0],r0
int16x4_t vset_lane_s16(int16_t value, int16x4_t vec, __constrange(0,3) int lane);
// VMOV.16 d0[0],r0
int32x2_t vset_lane_s32(int32_t value, int32x2_t vec, __constrange(0,1) int lane);
// VMOV.32 d0[0],r0
poly8x8_t vset_lane_p8(poly8_t value, poly8x8_t vec, __constrange(0,7) int lane);
// VMOV.8 d0[0],r0
poly16x4_t vset_lane_p16(poly16_t value, poly16x4_t vec, __constrange(0,3) int lane);
// VMOV.16 d0[0],r0
float32x2_t vset_lane_f32(float32_t value, float32x2_t vec, __constrange(0,1) int lane);
// VMOV.32 d0[0],r0
uint8x16_t vsetq_lane_u8(uint8_t value, uint8x16_t vec, __constrange(0,15) int lane);
// VMOV.8 d0[0],r0
uint16x8_t vsetq_lane_u16(uint16_t value, uint16x8_t vec, __constrange(0,7) int lane);
// VMOV.16 d0[0],r0
uint32x4_t vsetq_lane_u32(uint32_t value, uint32x4_t vec, __constrange(0,3) int lane);
// VMOV.32 d0[0],r0
int8x16_t vsetq_lane_s8(int8_t value, int8x16_t vec, __constrange(0,15) int lane);
// VMOV.8 d0[0],r0
int16x8_t vsetq_lane_s16(int16_t value, int16x8_t vec, __constrange(0,7) int lane);
// VMOV.16 d0[0],r0
int32x4_t vsetq_lane_s32(int32_t value, int32x4_t vec, __constrange(0,3) int lane);

```

```

// VMOV.32 d0[0],r0
poly8x16_t vsetq_lane_p8(poly8_t value, poly8x16_t vec, __constrange(0,15) int lane);
// VMOV.8 d0[0],r0
poly16x8_t vsetq_lane_p16(poly16_t value, poly16x8_t vec, __constrange(0,7) int lane);
// VMOV.16 d0[0],r0
float32x4_t vsetq_lane_f32(float32_t value, float32x4_t vec, __constrange(0,3) int lane);
// VMOV.32 d0[0],r0
int64x1_t vset_lane_s64(int64_t value, int64x1_t vec, __constrange(0,0) int lane);
// VMOV d0,r0,r0
uint64x1_t vset_lane_u64(uint64_t value, uint64x1_t vec, __constrange(0,0) int lane);
// VMOV d0,r0,r0
int64x2_t vsetq_lane_s64(int64_t value, int64x2_t vec, __constrange(0,1) int lane);
// VMOV d0,r0,r0
uint64x2_t vsetq_lane_u64(uint64_t value, uint64x2_t vec, __constrange(0,1) int lane);
// VMOV d0,r0,r0

```

E.3.18 Initialize a vector from bit pattern

These intrinsics create a vector from a literal bit pattern.

```

int8x8_t vcreate_s8(uint64_t a); // VMOV d0,r0,r0
int16x4_t vcreate_s16(uint64_t a); // VMOV d0,r0,r0
int32x2_t vcreate_s32(uint64_t a); // VMOV d0,r0,r0
float32x2_t vcreate_f32(uint64_t a); // VMOV d0,r0,r0
uint8x8_t vcreate_u8(uint64_t a); // VMOV d0,r0,r0
uint16x4_t vcreate_u16(uint64_t a); // VMOV d0,r0,r0
uint32x2_t vcreate_u32(uint64_t a); // VMOV d0,r0,r0
uint64x1_t vcreate_u64(uint64_t a); // VMOV d0,r0,r0
poly8x8_t vcreate_p8(uint64_t a); // VMOV d0,r0,r0
poly16x4_t vcreate_p16(uint64_t a); // VMOV d0,r0,r0
int64x1_t vcreate_s64(uint64_t a); // VMOV d0,r0,r0

```

E.3.19 Set all lanes to same value

These intrinsics set all lanes to the same value.

Set all lanes to the same value

```

uint8x8_t vdup_n_u8(uint8_t value); // VDUP.8 d0,r0
uint16x4_t vdup_n_u16(uint16_t value); // VDUP.16 d0,r0
uint32x2_t vdup_n_u32(uint32_t value); // VDUP.32 d0,r0
int8x8_t vdup_n_s8(int8_t value); // VDUP.8 d0,r0
int16x4_t vdup_n_s16(int16_t value); // VDUP.16 d0,r0
int32x2_t vdup_n_s32(int32_t value); // VDUP.32 d0,r0
poly8x8_t vdup_n_p8(poly8_t value); // VDUP.8 d0,r0
poly16x4_t vdup_n_p16(poly16_t value); // VDUP.16 d0,r0
float32x2_t vdup_n_f32(float32_t value); // VDUP.32 d0,r0

```

```

uint8x16_t vdupq_n_u8(uint8_t value);           // VDUP.8 q0,r0
uint16x8_t vdupq_n_u16(uint16_t value);         // VDUP.16 q0,r0
uint32x4_t vdupq_n_u32(uint32_t value);         // VDUP.32 q0,r0
int8x16_t vdupq_n_s8(int8_t value);             // VDUP.8 q0,r0
int16x8_t vdupq_n_s16(int16_t value);           // VDUP.16 q0,r0
int32x4_t vdupq_n_s32(int32_t value);           // VDUP.32 q0,r0
poly8x16_t vdupq_n_p8(poly8_t value);           // VDUP.8 q0,r0
poly16x8_t vdupq_n_p16(poly16_t value);         // VDUP.16 q0,r0
float32x4_t vdupq_n_f32(float32_t value);       // VDUP.32 q0,r0
int64x1_t vdup_n_s64(int64_t value);            // VMOV d0,r0,r0
uint64x1_t vdup_n_u64(uint64_t value);          // VMOV d0,r0,r0
int64x2_t vdupq_n_s64(int64_t value);           // VMOV d0,r0,r0
uint64x2_t vdupq_n_u64(uint64_t value);         // VMOV d0,r0,r0
uint8x8_t vmov_n_u8(uint8_t value);            // VDUP.8 d0,r0
uint16x4_t vmov_n_u16(uint16_t value);          // VDUP.16 d0,r0
uint32x2_t vmov_n_u32(uint32_t value);          // VDUP.32 d0,r0
int8x8_t vmov_n_s8(int8_t value);              // VDUP.8 d0,r0
int16x4_t vmov_n_s16(int16_t value);            // VDUP.16 d0,r0
int32x2_t vmov_n_s32(int32_t value);            // VDUP.32 d0,r0
poly8x8_t vmov_n_p8(poly8_t value);            // VDUP.8 d0,r0
poly16x4_t vmov_n_p16(poly16_t value);          // VDUP.16 d0,r0
float32x2_t vmov_n_f32(float32_t value);        // VDUP.32 d0,r0
uint8x16_t vmovq_n_u8(uint8_t value);           // VDUP.8 q0,r0
uint16x8_t vmovq_n_u16(uint16_t value);         // VDUP.16 q0,r0
uint32x4_t vmovq_n_u32(uint32_t value);         // VDUP.32 q0,r0
int8x16_t vmovq_n_s8(int8_t value);            // VDUP.8 q0,r0
int16x8_t vmovq_n_s16(int16_t value);           // VDUP.16 q0,r0
int32x4_t vmovq_n_s32(int32_t value);           // VDUP.32 q0,r0
poly8x16_t vmovq_n_p8(poly8_t value);           // VDUP.8 q0,r0
poly16x8_t vmovq_n_p16(poly16_t value);         // VDUP.16 q0,r0
float32x4_t vmovq_n_f32(float32_t value);       // VDUP.32 q0,r0
int64x1_t vmov_n_s64(int64_t value);            // VMOV d0,r0,r0
uint64x1_t vmov_n_u64(uint64_t value);          // VMOV d0,r0,r0
int64x2_t vmovq_n_s64(int64_t value);           // VMOV d0,r0,r0
uint64x2_t vmovq_n_u64(uint64_t value);         // VMOV d0,r0,r0

```

Set all lanes to the value of one lane of a vector

```

uint8x8_t vdup_lane_u8(uint8x8_t vec, __constrange(0,7) int lane); // VDUP.8 d0,d0[0]
uint16x4_t vdup_lane_u16(uint16x4_t vec, __constrange(0,3) int lane); // VDUP.16 d0,d0[0]
uint32x2_t vdup_lane_u32(uint32x2_t vec, __constrange(0,1) int lane); // VDUP.32 d0,d0[0]
int8x8_t vdup_lane_s8(int8x8_t vec, __constrange(0,7) int lane); // VDUP.8 d0,d0[0]
int16x4_t vdup_lane_s16(int16x4_t vec, __constrange(0,3) int lane); // VDUP.16 d0,d0[0]
int32x2_t vdup_lane_s32(int32x2_t vec, __constrange(0,1) int lane); // VDUP.32 d0,d0[0]
poly8x8_t vdup_lane_p8(poly8x8_t vec, __constrange(0,7) int lane); // VDUP.8 d0,d0[0]
poly16x4_t vdup_lane_p16(poly16x4_t vec, __constrange(0,3) int lane); // VDUP.16 d0,d0[0]
float32x2_t vdup_lane_f32(float32x2_t vec, __constrange(0,1) int lane); // VDUP.32 d0,d0[0]
uint8x16_t vdupq_lane_u8(uint8x8_t vec, __constrange(0,7) int lane); // VDUP.8 q0,d0[0]
uint16x8_t vdupq_lane_u16(uint16x4_t vec, __constrange(0,3) int lane); // VDUP.16 q0,d0[0]

```

```

uint32x4_t vdupq_lane_u32(uint32x2_t vec, __constrange(0,1) int lane); // VDUP.32 q0,d0[0]
int8x16_t vdupq_lane_s8(int8x8_t vec, __constrange(0,7) int lane); // VDUP.8 q0,d0[0]
int16x8_t vdupq_lane_s16(int16x4_t vec, __constrange(0,3) int lane); // VDUP.16 q0,d0[0]
int32x4_t vdupq_lane_s32(int32x2_t vec, __constrange(0,1) int lane); // VDUP.32 q0,d0[0]
poly8x16_t vdupq_lane_p8(poly8x8_t vec, __constrange(0,7) int lane); // VDUP.8 q0,d0[0]
poly16x8_t vdupq_lane_p16(poly16x4_t vec, __constrange(0,3) int lane); // VDUP.16 q0,d0[0]
float32x4_t vdupq_lane_f32(float32x2_t vec, __constrange(0,1) int lane); // VDUP.32 q0,d0[0]
int64x1_t vdup_lane_s64(int64x1_t vec, __constrange(0,0) int lane); // VMOV d0,d0
uint64x1_t vdup_lane_u64(uint64x1_t vec, __constrange(0,0) int lane); // VMOV d0,d0
int64x2_t vdupq_lane_s64(int64x1_t vec, __constrange(0,0) int lane); // VMOV q0,q0
uint64x2_t vdupq_lane_u64(uint64x1_t vec, __constrange(0,0) int lane); // VMOV q0,q0

```

E.3.20 Combining vectors

These intrinsics join two 64 bit vectors into a single 128bit vector.

```

int8x16_t vcombine_s8(int8x8_t low, int8x8_t high); // VMOV d0,d0
int16x8_t vcombine_s16(int16x4_t low, int16x4_t high); // VMOV d0,d0
int32x4_t vcombine_s32(int32x2_t low, int32x2_t high); // VMOV d0,d0
int64x2_t vcombine_s64(int64x1_t low, int64x1_t high); // VMOV d0,d0
float32x4_t vcombine_f32(float32x2_t low, float32x2_t high); // VMOV d0,d0
uint8x16_t vcombine_u8(uint8x8_t low, uint8x8_t high); // VMOV d0,d0
uint16x8_t vcombine_u16(uint16x4_t low, uint16x4_t high); // VMOV d0,d0
uint32x4_t vcombine_u32(uint32x2_t low, uint32x2_t high); // VMOV d0,d0
uint64x2_t vcombine_u64(uint64x1_t low, uint64x1_t high); // VMOV d0,d0
poly8x16_t vcombine_p8(poly8x8_t low, poly8x8_t high); // VMOV d0,d0
poly16x8_t vcombine_p16(poly16x4_t low, poly16x4_t high); // VMOV d0,d0

```

E.3.21 Splitting vectors

These intrinsics split a 128 bit vector into 2 component 64 bit vectors

```

int8x8_t vget_high_s8(int8x16_t a); // VMOV d0,d0
int16x4_t vget_high_s16(int16x8_t a); // VMOV d0,d0
int32x2_t vget_high_s32(int32x4_t a); // VMOV d0,d0
int64x1_t vget_high_s64(int64x2_t a); // VMOV d0,d0
float32x2_t vget_high_f32(float32x4_t a); // VMOV d0,d0
uint8x8_t vget_high_u8(uint8x16_t a); // VMOV d0,d0
uint16x4_t vget_high_u16(uint16x8_t a); // VMOV d0,d0
uint32x2_t vget_high_u32(uint32x4_t a); // VMOV d0,d0
uint64x1_t vget_high_u64(uint64x2_t a); // VMOV d0,d0
poly8x8_t vget_high_p8(poly8x16_t a); // VMOV d0,d0
poly16x4_t vget_high_p16(poly16x8_t a); // VMOV d0,d0
int8x8_t vget_low_s8(int8x16_t a); // VMOV d0,d0
int16x4_t vget_low_s16(int16x8_t a); // VMOV d0,d0
int32x2_t vget_low_s32(int32x4_t a); // VMOV d0,d0
int64x1_t vget_low_s64(int64x2_t a); // VMOV d0,d0
float32x2_t vget_low_f32(float32x4_t a); // VMOV d0,d0
uint8x8_t vget_low_u8(uint8x16_t a); // VMOV d0,d0

```

```

uint16x4_t vget_low_u16(uint16x8_t a);    // VMOV d0,d0
uint32x2_t vget_low_u32(uint32x4_t a);    // VMOV d0,d0
uint64x1_t vget_low_u64(uint64x2_t a);    // VMOV d0,d0
poly8x8_t  vget_low_p8(poly8x16_t a);    // VMOV d0,d0
poly16x4_t vget_low_p16(poly16x8_t a);    // VMOV d0,d0

```

E.3.22 Converting vectors

These intrinsics are used to convert vectors.

Convert from float

```

int32x2_t vcvt_s32_f32(float32x2_t a);    // VCVT.S32.F32 d0, d0
uint32x2_t vcvt_u32_f32(float32x2_t a);    // VCVT.U32.F32 d0, d0
int32x4_t vcvtq_s32_f32(float32x4_t a);    // VCVT.S32.F32 q0, q0
uint32x4_t vcvtq_u32_f32(float32x4_t a);    // VCVT.U32.F32 q0, q0
int32x2_t vcvt_n_s32_f32(float32x2_t a, __constrange(1,32) int b); // VCVT.S32.F32 d0, d0, #32
uint32x2_t vcvt_n_u32_f32(float32x2_t a, __constrange(1,32) int b); // VCVT.U32.F32 d0, d0, #32
int32x4_t vcvtq_n_s32_f32(float32x4_t a, __constrange(1,32) int b); // VCVT.S32.F32 q0, q0, #32
uint32x4_t vcvtq_n_u32_f32(float32x4_t a, __constrange(1,32) int b); // VCVT.U32.F32 q0, q0, #32

```

Convert to float

```

float32x2_t vcvt_f32_s32(int32x2_t a);    // VCVT.F32.S32 d0, d0
float32x2_t vcvt_f32_u32(uint32x2_t a);    // VCVT.F32.U32 d0, d0
float32x4_t vcvtq_f32_s32(int32x4_t a);    // VCVT.F32.S32 q0, q0
float32x4_t vcvtq_f32_u32(uint32x4_t a);    // VCVT.F32.U32 q0, q0
float32x2_t vcvt_n_f32_s32(int32x2_t a, __constrange(1,32) int b); // VCVT.F32.S32 d0, d0, #32
float32x2_t vcvt_n_f32_u32(uint32x2_t a, __constrange(1,32) int b); // VCVT.F32.U32 d0, d0, #32
float32x4_t vcvtq_n_f32_s32(int32x4_t a, __constrange(1,32) int b); // VCVT.F32.S32 q0, q0, #32
float32x4_t vcvtq_n_f32_u32(uint32x4_t a, __constrange(1,32) int b); // VCVT.F32.U32 q0, q0, #32

```

Vector narrow integer

```

int8x8_t  vmovn_s16(int16x8_t a);    // VMOVN.I16 d0,q0
int16x4_t vmovn_s32(int32x4_t a);    // VMOVN.I32 d0,q0
int32x2_t vmovn_s64(int64x2_t a);    // VMOVN.I64 d0,q0
uint8x8_t vmovn_u16(uint16x8_t a);    // VMOVN.I16 d0,q0
uint16x4_t vmovn_u32(uint32x4_t a);    // VMOVN.I32 d0,q0
uint32x2_t vmovn_u64(uint64x2_t a);    // VMOVN.I64 d0,q0

```

Vector long move

```

int16x8_t vmovl_s8(int8x8_t a);    // VMOVL.S8 q0,d0
int32x4_t vmovl_s16(int16x4_t a);    // VMOVL.S16 q0,d0
int64x2_t vmovl_s32(int32x2_t a);    // VMOVL.S32 q0,d0

```

```
uint16x8_t vmovl_u8(uint8x8_t a);    // VMOVL.U8 q0,d0
uint32x4_t vmovl_u16(uint16x4_t a);  // VMOVL.U16 q0,d0
uint64x2_t vmovl_u32(uint32x2_t a);  // VMOVL.U32 q0,d0
```

Vector saturating narrow integer

```
int8x8_t  vqmovn_s16(int16x8_t a);  // VQMOVN.S16 d0,q0
int16x4_t vqmovn_s32(int32x4_t a);  // VQMOVN.S32 d0,q0
int32x2_t vqmovn_s64(int64x2_t a);  // VQMOVN.S64 d0,q0
uint8x8_t vqmovn_u16(uint16x8_t a); // VQMOVN.U16 d0,q0
uint16x4_t vqmovn_u32(uint32x4_t a); // VQMOVN.U32 d0,q0
uint32x2_t vqmovn_u64(uint64x2_t a); // VQMOVN.U64 d0,q0
```

Vector saturating narrow integer signed->unsigned

```
uint8x8_t vqmovun_s16(int16x8_t a); // VQMOVUN.S16 d0,q0
uint16x4_t vqmovun_s32(int32x4_t a); // VQMOVUN.S32 d0,q0
uint32x2_t vqmovun_s64(int64x2_t a); // VQMOVUN.S64 d0,q0
```

E.3.23 Table look up

```
uint8x8_t vtbl1_u8(uint8x8_t a, uint8x8_t b); // VTBL.8 d0, {d0}, d0
int8x8_t  vtbl1_s8(int8x8_t a, int8x8_t b);   // VTBL.8 d0, {d0}, d0
poly8x8_t vtbl1_p8(poly8x8_t a, uint8x8_t b); // VTBL.8 d0, {d0}, d0
uint8x8_t vtbl2_u8(uint8x8x2_t a, uint8x8_t b); // VTBL.8 d0, {d0, d1}, d0
int8x8_t  vtbl2_s8(int8x8x2_t a, int8x8_t b);   // VTBL.8 d0, {d0, d1}, d0
poly8x8_t vtbl2_p8(poly8x8x2_t a, uint8x8_t b); // VTBL.8 d0, {d0, d1}, d0
uint8x8_t vtbl3_u8(uint8x8x3_t a, uint8x8_t b); // VTBL.8 d0, {d0, d1, d2}, d0
int8x8_t  vtbl3_s8(int8x8x3_t a, int8x8_t b);   // VTBL.8 d0, {d0, d1, d2}, d0
poly8x8_t vtbl3_p8(poly8x8x3_t a, uint8x8_t b); // VTBL.8 d0, {d0, d1, d2}, d0
uint8x8_t vtbl4_u8(uint8x8x4_t a, uint8x8_t b); // VTBL.8 d0, {d0, d1, d2, d3}, d0
int8x8_t  vtbl4_s8(int8x8x4_t a, int8x8_t b);   // VTBL.8 d0, {d0, d1, d2, d3}, d0
poly8x8_t vtbl4_p8(poly8x8x4_t a, uint8x8_t b); // VTBL.8 d0, {d0, d1, d2, d3}, d0
```

E.3.24 Extended table look up intrinsics

```
uint8x8_t vtbx1_u8(uint8x8_t a, uint8x8_t b, uint8x8_t c); // VTBX.8 d0, {d0}, d0
int8x8_t  vtbx1_s8(int8x8_t a, int8x8_t b, int8x8_t c);    // VTBX.8 d0, {d0}, d0
poly8x8_t vtbx1_p8(poly8x8_t a, poly8x8_t b, uint8x8_t c); // VTBX.8 d0, {d0}, d0
uint8x8_t vtbx2_u8(uint8x8_t a, uint8x8x2_t b, uint8x8_t c); // VTBX.8 d0, {d0, d1}, d0
int8x8_t  vtbx2_s8(int8x8_t a, int8x8x2_t b, int8x8_t c);    // VTBX.8 d0, {d0, d1}, d0
poly8x8_t vtbx2_p8(poly8x8_t a, poly8x8x2_t b, uint8x8_t c); // VTBX.8 d0, {d0, d1}, d0
uint8x8_t vtbx3_u8(uint8x8_t a, uint8x8x3_t b, uint8x8_t c); // VTBX.8 d0, {d0, d1, d2}, d0
int8x8_t  vtbx3_s8(int8x8_t a, int8x8x3_t b, int8x8_t c);    // VTBX.8 d0, {d0, d1, d2}, d0
poly8x8_t vtbx3_p8(poly8x8_t a, poly8x8x3_t b, uint8x8_t c); // VTBX.8 d0, {d0, d1, d2}, d0
uint8x8_t vtbx4_u8(uint8x8_t a, uint8x8x4_t b, uint8x8_t c); // VTBX.8 d0, {d0, d1, d2, d3}, d0
int8x8_t  vtbx4_s8(int8x8_t a, int8x8x4_t b, int8x8_t c);    // VTBX.8 d0, {d0, d1, d2, d3}, d0
poly8x8_t vtbx4_p8(poly8x8_t a, poly8x8x4_t b, uint8x8_t c); // VTBX.8 d0, {d0, d1, d2, d3}, d0
```

E.3.25 Operations with a scalar value

Efficient code generation for these intrinsics is only guaranteed when the scalar argument is either a constant or a use of one of the `vget_lane` intrinsics.

Vector multiply accumulate with scalar

```
int16x4_t  vmla_lane_s16(int16x4_t a, int16x4_t b, int16x4_t v, __constrange(0,3) int l);
// VMLA.I16 d0, d0, d0[0]
int32x2_t  vmla_lane_s32(int32x2_t a, int32x2_t b, int32x2_t v, __constrange(0,1) int l);
// VMLA.I32 d0, d0, d0[0]
uint16x4_t vmla_lane_u16(uint16x4_t a, uint16x4_t b, uint16x4_t v, __constrange(0,3) int l);
// VMLA.I16 d0, d0, d0[0]
uint32x2_t vmla_lane_u32(uint32x2_t a, uint32x2_t b, uint32x2_t v, __constrange(0,1) int l);
// VMLA.I32 d0, d0, d0[0]
float32x2_t vmla_lane_f32(float32x2_t a, float32x2_t b, float32x2_t v, __constrange(0,1) int l);
// VMLA.F32 d0, d0, d0[0]
int16x8_t  vmlaq_lane_s16(int16x8_t a, int16x8_t b, int16x4_t v, __constrange(0,3) int l);
// VMLA.I16 q0, q0, d0[0]
int32x4_t  vmlaq_lane_s32(int32x4_t a, int32x4_t b, int32x2_t v, __constrange(0,1) int l);
// VMLA.I32 q0, q0, d0[0]
uint16x8_t vmlaq_lane_u16(uint16x8_t a, uint16x8_t b, uint16x4_t v, __constrange(0,3) int l);
// VMLA.I16 q0, q0, d0[0]
uint32x4_t vmlaq_lane_u32(uint32x4_t a, uint32x4_t b, uint32x2_t v, __constrange(0,1) int l);
// VMLA.I32 q0, q0, d0[0]
float32x4_t vmlaq_lane_f32(float32x4_t a, float32x4_t b, float32x2_t v, __constrange(0,1) int l);
// VMLA.F32 q0, q0, d0[0]
```

Vector widening multiply accumulate with scalar

```
int32x4_t  vmlal_lane_s16(int32x4_t a, int16x4_t b, int16x4_t v, __constrange(0,3) int l);
// VMLAL.S16 q0, d0, d0[0]
int64x2_t  vmlal_lane_s32(int64x2_t a, int32x2_t b, int32x2_t v, __constrange(0,1) int l);
// VMLAL.S32 q0, d0, d0[0]
uint32x4_t vmlal_lane_u16(uint32x4_t a, uint16x4_t b, uint16x4_t v, __constrange(0,3) int l);
// VMLAL.U16 q0, d0, d0[0]
uint64x2_t vmlal_lane_u32(uint64x2_t a, uint32x2_t b, uint32x2_t v, __constrange(0,1) int l);
// VMLAL.U32 q0, d0, d0[0]
```

Vector widening saturating doubling multiply accumulate with scalar

```
int32x4_t  vqdmmlal_lane_s16(int32x4_t a, int16x4_t b, int16x4_t v, __constrange(0,3) int l);
// VQDMLAL.S16 q0, d0, d0[0]
int64x2_t  vqdmmlal_lane_s32(int64x2_t a, int32x2_t b, int32x2_t v, __constrange(0,1) int l);
// VQDMLAL.S32 q0, d0, d0[0]
```

Vector multiply subtract with scalar

```

int16x4_t  vmls_lane_s16(int16x4_t a, int16x4_t b, int16x4_t v, __constrange(0,3) int l);
// VMLS.I16 d0, d0, d0[0]
int32x2_t  vmls_lane_s32(int32x2_t a, int32x2_t b, int32x2_t v, __constrange(0,1) int l);
// VMLS.I32 d0, d0, d0[0]
uint16x4_t vmls_lane_u16(uint16x4_t a, uint16x4_t b, uint16x4_t v, __constrange(0,3) int l);
// VMLS.I16 d0, d0, d0[0]
uint32x2_t vmls_lane_u32(uint32x2_t a, uint32x2_t b, uint32x2_t v, __constrange(0,1) int l);
// VMLS.I32 d0, d0, d0[0]
float32x2_t vmls_lane_f32(float32x2_t a, float32x2_t b, float32x2_t v, __constrange(0,1) int l);
// VMLS.F32 d0, d0, d0[0]
int16x8_t  vmlsq_lane_s16(int16x8_t a, int16x8_t b, int16x4_t v, __constrange(0,3) int l);
// VMLS.I16 q0, q0, d0[0]
int32x4_t  vmlsq_lane_s32(int32x4_t a, int32x4_t b, int32x2_t v, __constrange(0,1) int l);
// VMLS.I32 q0, q0, d0[0]
uint16x8_t vmlsq_lane_u16(uint16x8_t a, uint16x8_t b, uint16x4_t v, __constrange(0,3) int l);
// VMLS.I16 q0, q0, d0[0]
uint32x4_t vmlsq_lane_u32(uint32x4_t a, uint32x4_t b, uint32x2_t v, __constrange(0,1) int l);
// VMLS.I32 q0, q0, d0[0]
float32x4_t vmlsq_lane_f32(float32x4_t a, float32x4_t b, float32x2_t v, __constrange(0,1) int l);
// VMLS.F32 q0, q0, d0[0]

```

Vector widening multiply subtract with scalar

```

int32x4_t  vmlsl_lane_s16(int32x4_t a, int16x4_t b, int16x4_t v, __constrange(0,3) int l);
// VMLSL.S16 q0, d0, d0[0]
int64x2_t  vmlsl_lane_s32(int64x2_t a, int32x2_t b, int32x2_t v, __constrange(0,1) int l);
// VMLSL.S32 q0, d0, d0[0]
uint32x4_t vmlsl_lane_u16(uint32x4_t a, uint16x4_t b, uint16x4_t v, __constrange(0,3) int l);
// VMLSL.U16 q0, d0, d0[0]
uint64x2_t vmlsl_lane_u32(uint64x2_t a, uint32x2_t b, uint32x2_t v, __constrange(0,1) int l);
// VMLSL.U32 q0, d0, d0[0]

```

Vector widening saturating doubling multiply subtract with scalar

```

int32x4_t  vqdm1sl_lane_s16(int32x4_t a, int16x4_t b, int16x4_t v, __constrange(0,3) int l);
// VQDMLSL.S16 q0, d0, d0[0]
int64x2_t  vqdm1sl_lane_s32(int64x2_t a, int32x2_t b, int32x2_t v, __constrange(0,1) int l);
// VQDMLSL.S32 q0, d0, d0[0]

```

Vector multiply by scalar

```

int16x4_t  vmul_n_s16(int16x4_t a, int16_t b); // VMUL.I16 d0,d0,d0[0]
int32x2_t  vmul_n_s32(int32x2_t a, int32_t b); // VMUL.I32 d0,d0,d0[0]
float32x2_t vmul_n_f32(float32x2_t a, float32_t b); // VMUL.F32 d0,d0,d0[0]
uint16x4_t vmul_n_u16(uint16x4_t a, uint16_t b); // VMUL.I16 d0,d0,d0[0]
uint32x2_t vmul_n_u32(uint32x2_t a, uint32_t b); // VMUL.I32 d0,d0,d0[0]

```

```

int16x8_t  vmulq_n_s16(int16x8_t a, int16_t b);    // VMUL.I16 q0,q0,d0[0]
int32x4_t  vmulq_n_s32(int32x4_t a, int32_t b);    // VMUL.I32 q0,q0,d0[0]
float32x4_t vmulq_n_f32(float32x4_t a, float32_t b); // VMUL.F32 q0,q0,d0[0]
uint16x8_t vmulq_n_u16(uint16x8_t a, uint16_t b);  // VMUL.I16 q0,q0,d0[0]
uint32x4_t vmulq_n_u32(uint32x4_t a, uint32_t b);  // VMUL.I32 q0,q0,d0[0]

```

Vector long multiply with scalar

```

int32x4_t vmull_n_s16(int16x4_t vec1, int16_t val2); // VMULL.S16 q0,d0,d0[0]
int64x2_t vmull_n_s32(int32x2_t vec1, int32_t val2); // VMULL.S32 q0,d0,d0[0]
uint32x4_t vmull_n_u16(uint16x4_t vec1, uint16_t val2); // VMULL.U16 q0,d0,d0[0]
uint64x2_t vmull_n_u32(uint32x2_t vec1, uint32_t val2); // VMULL.U32 q0,d0,d0[0]

```

Vector long multiply by scalar

```

int32x4_t vmull_lane_s16(int16x4_t vec1, int16x4_t val2, __constrange(0, 3) int val3);
// VMULL.S16 q0,d0,d0[0]
int64x2_t vmull_lane_s32(int32x2_t vec1, int32x2_t val2, __constrange(0, 1) int val3);
// VMULL.S32 q0,d0,d0[0]
uint32x4_t vmull_lane_u16(uint16x4_t vec1, uint16x4_t val2, __constrange(0, 3) int val3);
// VMULL.U16 q0,d0,d0[0]
uint64x2_t vmull_lane_u32(uint32x2_t vec1, uint32x2_t val2, __constrange(0, 1) int val3);
// VMULL.U32 q0,d0,d0[0]

```

Vector saturating doubling long multiply with scalar

```

int32x4_t vqdmull_n_s16(int16x4_t vec1, int16_t val2); // VQDMULL.S16 q0,d0,d0[0]
int64x2_t vqdmull_n_s32(int32x2_t vec1, int32_t val2); // VQDMULL.S32 q0,d0,d0[0]

```

Vector saturating doubling long multiply by scalar

```

int32x4_t vqdmull_lane_s16(int16x4_t vec1, int16x4_t val2, __constrange(0, 3) int val3);
// VQDMULL.S16 q0,d0,d0[0]
int64x2_t vqdmull_lane_s32(int32x2_t vec1, int32x2_t val2, __constrange(0, 1) int val3);
// VQDMULL.S32 q0,d0,d0[0]

```

Vector saturating doubling multiply high with scalar

```

int16x4_t vqdmulh_n_s16(int16x4_t vec1, int16_t val2); // VQDMULH.S16 d0,d0,d0[0]
int32x2_t vqdmulh_n_s32(int32x2_t vec1, int32_t val2); // VQDMULH.S32 d0,d0,d0[0]
int16x8_t vqdmulhq_n_s16(int16x8_t vec1, int16_t val2); // VQDMULH.S16 q0,q0,d0[0]
int32x4_t vqdmulhq_n_s32(int32x4_t vec1, int32_t val2); // VQDMULH.S32 q0,q0,d0[0]

```

Vector saturating doubling multiply high by scalar

```

int16x4_t vqdmulh_lane_s16(int16x4_t vec1, int16x4_t val2, __constrange(0, 3) int val3);
// VQDMULH.S16 d0,d0,d0[0]

```

```

int32x2_t vqdmulh_lane_s32(int32x2_t vec1, int32x2_t val2, __constrange(0, 1) int val3);
// VQDMULH.S32 d0,d0,d0[0]
int16x8_t vqdmulhq_lane_s16(int16x8_t vec1, int16x4_t val2, __constrange(0, 3) int val3);
// VQDMULH.S16 q0,q0,d0[0]
int32x4_t vqdmulhq_lane_s32(int32x4_t vec1, int32x2_t val2, __constrange(0, 1) int val3);
// VQDMULH.S32 q0,q0,d0[0]

```

Vector saturating rounding doubling multiply high with scalar

```

int16x4_t vqrdmulh_n_s16(int16x4_t vec1, int16_t val2); // VQRDMULH.S16 d0,d0,d0[0]
int32x2_t vqrdmulh_n_s32(int32x2_t vec1, int32_t val2); // VQRDMULH.S32 d0,d0,d0[0]
int16x8_t vqrdmulhq_n_s16(int16x8_t vec1, int16_t val2); // VQRDMULH.S16 q0,q0,d0[0]
int32x4_t vqrdmulhq_n_s32(int32x4_t vec1, int32_t val2); // VQRDMULH.S32 q0,q0,d0[0]

```

Vector rounding saturating doubling multiply high by scalar

```

int16x4_t vqrdmulh_lane_s16(int16x4_t vec1, int16x4_t val2, __constrange(0, 3) int val3);
// VQRDMULH.S16 d0,d0,d0[0]
int32x2_t vqrdmulh_lane_s32(int32x2_t vec1, int32x2_t val2, __constrange(0, 1) int val3);
// VQRDMULH.S32 d0,d0,d0[0]
int16x8_t vqrdmulhq_lane_s16(int16x8_t vec1, int16x4_t val2, __constrange(0, 3) int val3);
// VQRDMULH.S16 q0,q0,d0[0]
int32x4_t vqrdmulhq_lane_s32(int32x4_t vec1, int32x2_t val2, __constrange(0, 1) int val3);
// VQRDMULH.S32 q0,q0,d0[0]

```

Vector multiply accumulate with scalar

```

int16x4_t vmla_n_s16(int16x4_t a, int16x4_t b, int16_t c); // VMLA.I16 d0, d0, d0[0]
int32x2_t vmla_n_s32(int32x2_t a, int32x2_t b, int32_t c); // VMLA.I32 d0, d0, d0[0]
uint16x4_t vmla_n_u16(uint16x4_t a, uint16x4_t b, uint16_t c); // VMLA.I16 d0, d0, d0[0]
uint32x2_t vmla_n_u32(uint32x2_t a, uint32x2_t b, uint32_t c); // VMLA.I32 d0, d0, d0[0]
float32x2_t vmla_n_f32(float32x2_t a, float32x2_t b, float32_t c); // VMLA.F32 d0, d0, d0[0]
int16x8_t vmlaq_n_s16(int16x8_t a, int16x8_t b, int16_t c); // VMLA.I16 q0, q0, d0[0]
int32x4_t vmlaq_n_s32(int32x4_t a, int32x4_t b, int32_t c); // VMLA.I32 q0, q0, d0[0]
uint16x8_t vmlaq_n_u16(uint16x8_t a, uint16x8_t b, uint16_t c); // VMLA.I16 q0, q0, d0[0]
uint32x4_t vmlaq_n_u32(uint32x4_t a, uint32x4_t b, uint32_t c); // VMLA.I32 q0, q0, d0[0]
float32x4_t vmlaq_n_f32(float32x4_t a, float32x4_t b, float32_t c); // VMLA.F32 q0, q0, d0[0]

```

Vector widening multiply accumulate with scalar

```

int32x4_t vmlal_n_s16(int32x4_t a, int16x4_t b, int16_t c); // VMLAL.S16 q0, d0, d0[0]
int64x2_t vmlal_n_s32(int64x2_t a, int32x2_t b, int32_t c); // VMLAL.S32 q0, d0, d0[0]
uint32x4_t vmlal_n_u16(uint32x4_t a, uint16x4_t b, uint16_t c); // VMLAL.U16 q0, d0, d0[0]
uint64x2_t vmlal_n_u32(uint64x2_t a, uint32x2_t b, uint32_t c); // VMLAL.U32 q0, d0, d0[0]

```

Vector widening saturating doubling multiply accumulate with scalar

```
int32x4_t  vqdmmlal_n_s16(int32x4_t a, int16x4_t b, int16_t c);    // VQDMLAL.S16 q0, d0, d0[0]
int64x2_t  vqdmmlal_n_s32(int64x2_t a, int32x2_t b, int32_t c);    // VQDMLAL.S32 q0, d0, d0[0]
```

Vector multiply subtract with scalar

```
int16x4_t  vmls_n_s16(int16x4_t a, int16x4_t b, int16_t c);        // VMLS.I16 d0, d0, d0[0]
int32x2_t  vmls_n_s32(int32x2_t a, int32x2_t b, int32_t c);        // VMLS.I32 d0, d0, d0[0]
uint16x4_t vmls_n_u16(uint16x4_t a, uint16x4_t b, uint16_t c);     // VMLS.I16 d0, d0, d0[0]
uint32x2_t vmls_n_u32(uint32x2_t a, uint32x2_t b, uint32_t c);     // VMLS.I32 d0, d0, d0[0]
float32x2_t vmls_n_f32(float32x2_t a, float32x2_t b, float32_t c); // VMLS.F32 d0, d0, d0[0]
int16x8_t  vmlsq_n_s16(int16x8_t a, int16x8_t b, int16_t c);       // VMLS.I16 q0, q0, d0[0]
int32x4_t  vmlsq_n_s32(int32x4_t a, int32x4_t b, int32_t c);       // VMLS.I32 q0, q0, d0[0]
uint16x8_t vmlsq_n_u16(uint16x8_t a, uint16x8_t b, uint16_t c);    // VMLS.I16 q0, q0, d0[0]
uint32x4_t vmlsq_n_u32(uint32x4_t a, uint32x4_t b, uint32_t c);    // VMLS.I32 q0, q0, d0[0]
float32x4_t vmlsq_n_f32(float32x4_t a, float32x4_t b, float32_t c); // VMLS.F32 q0, q0, d0[0]
```

Vector widening multiply subtract with scalar

```
int32x4_t  vmlsl_n_s16(int32x4_t a, int16x4_t b, int16_t c);       // VMLSL.S16 q0, d0, d0[0]
int64x2_t  vmlsl_n_s32(int64x2_t a, int32x2_t b, int32_t c);       // VMLSL.S32 q0, d0, d0[0]
uint32x4_t vmlsl_n_u16(uint32x4_t a, uint16x4_t b, uint16_t c);    // VMLSL.U16 q0, d0, d0[0]
uint64x2_t vmlsl_n_u32(uint64x2_t a, uint32x2_t b, uint32_t c);    // VMLSL.U32 q0, d0, d0[0]
```

Vector widening saturating doubling multiply subtract with scalar

```
int32x4_t  vqdmmlsl_n_s16(int32x4_t a, int16x4_t b, int16_t c);    // VQDMLSL.S16 q0, d0, d0[0]
int64x2_t  vqdmmlsl_n_s32(int64x2_t a, int32x2_t b, int32_t c);    // VQDMLSL.S32 q0, d0, d0[0]
```

E.3.26 Vector extract

```
int8x8_t   vext_s8(int8x8_t a, int8x8_t b, __constrange(0,7) int c); // VEXT.8 d0,d0,d0,#0
uint8x8_t  vext_u8(uint8x8_t a, uint8x8_t b, __constrange(0,7) int c); // VEXT.8 d0,d0,d0,#0
poly8x8_t  vext_p8(poly8x8_t a, poly8x8_t b, __constrange(0,7) int c); // VEXT.8 d0,d0,d0,#0
int16x4_t  vext_s16(int16x4_t a, int16x4_t b, __constrange(0,3) int c); // VEXT.16 d0,d0,d0,#0
uint16x4_t vext_u16(uint16x4_t a, uint16x4_t b, __constrange(0,3) int c); // VEXT.16 d0,d0,d0,#0
poly16x4_t vext_p16(poly16x4_t a, poly16x4_t b, __constrange(0,3) int c); // VEXT.16 d0,d0,d0,#0
int32x2_t  vext_s32(int32x2_t a, int32x2_t b, __constrange(0,1) int c); // VEXT.32 d0,d0,d0,#0
uint32x2_t vext_u32(uint32x2_t a, uint32x2_t b, __constrange(0,1) int c); // VEXT.32 d0,d0,d0,#0
int64x1_t  vext_s64(int64x1_t a, int64x1_t b, __constrange(0,0) int c); // VEXT.64 d0,d0,d0,#0
uint64x1_t vext_u64(uint64x1_t a, uint64x1_t b, __constrange(0,0) int c); // VEXT.64 d0,d0,d0,#0
int8x16_t  vextq_s8(int8x16_t a, int8x16_t b, __constrange(0,15) int c); // VEXT.8 q0,q0,q0,#0
uint8x16_t vextq_u8(uint8x16_t a, uint8x16_t b, __constrange(0,15) int c); // VEXT.8 q0,q0,q0,#0
poly8x16_t vextq_p8(poly8x16_t a, poly8x16_t b, __constrange(0,15) int c); // VEXT.8 q0,q0,q0,#0
int16x8_t  vextq_s16(int16x8_t a, int16x8_t b, __constrange(0,7) int c); // VEXT.16 q0,q0,q0,#0
uint16x8_t vextq_u16(uint16x8_t a, uint16x8_t b, __constrange(0,7) int c); // VEXT.16 q0,q0,q0,#0
poly16x8_t vextq_p16(poly16x8_t a, poly16x8_t b, __constrange(0,7) int c); // VEXT.16 q0,q0,q0,#0
```

```

int32x4_t vextq_s32(int32x4_t a, int32x4_t b, __constrange(0,3) int c); // VEXT.32 q0,q0,q0,#0
uint32x4_t vextq_u32(uint32x4_t a, uint32x4_t b, __constrange(0,3) int c); // VEXT.32 q0,q0,q0,#0
int64x2_t vextq_s64(int64x2_t a, int64x2_t b, __constrange(0,1) int c); // VEXT.64 q0,q0,q0,#0
uint64x2_t vextq_u64(uint64x2_t a, uint64x2_t b, __constrange(0,1) int c); // VEXT.64 q0,q0,q0,#0

```

E.3.27 Reverse vector elements (swap endianness)

VREVn.m reverses the order of the m-bit lanes within a set that is n bits wide.

```

int8x8_t      vrev64_s8(int8x8_t vec);      // VREV64.8 d0,d0
int16x4_t     vrev64_s16(int16x4_t vec);     // VREV64.16 d0,d0
int32x2_t     vrev64_s32(int32x2_t vec);     // VREV64.32 d0,d0
uint8x8_t     vrev64_u8(uint8x8_t vec);      // VREV64.8 d0,d0
uint16x4_t    vrev64_u16(uint16x4_t vec);    // VREV64.16 d0,d0
uint32x2_t    vrev64_u32(uint32x2_t vec);    // VREV64.32 d0,d0
poly8x8_t     vrev64_p8(poly8x8_t vec);      // VREV64.8 d0,d0
poly16x4_t    vrev64_p16(poly16x4_t vec);    // VREV64.16 d0,d0
float32x2_t   vrev64_f32(float32x2_t vec);   // VREV64.32 d0,d0
int8x16_t     vrev64q_s8(int8x16_t vec);     // VREV64.8 q0,q0
int16x8_t     vrev64q_s16(int16x8_t vec);     // VREV64.16 q0,q0
int32x4_t     vrev64q_s32(int32x4_t vec);     // VREV64.32 q0,q0
uint8x16_t    vrev64q_u8(uint8x16_t vec);     // VREV64.8 q0,q0
uint16x8_t    vrev64q_u16(uint16x8_t vec);    // VREV64.16 q0,q0
uint32x4_t    vrev64q_u32(uint32x4_t vec);    // VREV64.32 q0,q0
poly8x16_t    vrev64q_p8(poly8x16_t vec);     // VREV64.8 q0,q0
poly16x8_t    vrev64q_p16(poly16x8_t vec);    // VREV64.16 q0,q0
float32x4_t   vrev64q_f32(float32x4_t vec);   // VREV64.32 q0,q0
int8x8_t      vrev32_s8(int8x8_t vec);      // VREV32.8 d0,d0
int16x4_t     vrev32_s16(int16x4_t vec);     // VREV32.16 d0,d0
uint8x8_t     vrev32_u8(uint8x8_t vec);      // VREV32.8 d0,d0
uint16x4_t    vrev32_u16(uint16x4_t vec);    // VREV32.16 d0,d0
poly8x8_t     vrev32_p8(poly8x8_t vec);      // VREV32.8 d0,d0
int8x16_t     vrev32q_s8(int8x16_t vec);     // VREV32.8 q0,q0
int16x8_t     vrev32q_s16(int16x8_t vec);     // VREV32.16 q0,q0
uint8x16_t    vrev32q_u8(uint8x16_t vec);     // VREV32.8 q0,q0
uint16x8_t    vrev32q_u16(uint16x8_t vec);    // VREV32.16 q0,q0
poly8x16_t    vrev32q_p8(poly8x16_t vec);     // VREV32.8 q0,q0
int8x8_t      vrev16_s8(int8x8_t vec);      // VREV16.8 d0,d0
uint8x8_t     vrev16_u8(uint8x8_t vec);      // VREV16.8 d0,d0
poly8x8_t     vrev16_p8(poly8x8_t vec);      // VREV16.8 d0,d0
int8x16_t     vrev16q_s8(int8x16_t vec);     // VREV16.8 q0,q0
uint8x16_t    vrev16q_u8(uint8x16_t vec);     // VREV16.8 q0,q0
poly8x16_t    vrev16q_p8(poly8x16_t vec);     // VREV16.8 q0,q0

```

E.3.28 Other single operand arithmetic

These intrinsics provide other single operand arithmetic.

Absolute: $Vd[i] = |Va[i]|$

```

int8x8_t   vabs_s8(int8x8_t a);      // VABS.S8 d0,d0
int16x4_t  vabs_s16(int16x4_t a);    // VABS.S16 d0,d0
int32x2_t  vabs_s32(int32x2_t a);    // VABS.S32 d0,d0
float32x2_t vabs_f32(float32x2_t a); // VABS.F32 d0,d0
int8x16_t  vabsq_s8(int8x16_t a);    // VABS.S8 q0,q0
int16x8_t  vabsq_s16(int16x8_t a);   // VABS.S16 q0,q0
int32x4_t  vabsq_s32(int32x4_t a);   // VABS.S32 q0,q0
float32x4_t vabsq_f32(float32x4_t a); // VABS.F32 q0,q0

```

Saturating absolute: $Vd[i] = \text{sat}(|Va[i]|)$

```

int8x8_t   vqabs_s8(int8x8_t a);      // VQABS.S8 d0,d0
int16x4_t  vqabs_s16(int16x4_t a);    // VQABS.S16 d0,d0
int32x2_t  vqabs_s32(int32x2_t a);    // VQABS.S32 d0,d0
int8x16_t  vqabsq_s8(int8x16_t a);    // VQABS.S8 q0,q0
int16x8_t  vqabsq_s16(int16x8_t a);   // VQABS.S16 q0,q0
int32x4_t  vqabsq_s32(int32x4_t a);   // VQABS.S32 q0,q0

```

Negate: $Vd[i] = -Va[i]$

```

int8x8_t   vneg_s8(int8x8_t a);      // VNEG.S8 d0,d0
int16x4_t  vneg_s16(int16x4_t a);    // VNEG.S16 d0,d0
int32x2_t  vneg_s32(int32x2_t a);    // VNEG.S32 d0,d0
float32x2_t vneg_f32(float32x2_t a); // VNEG.F32 d0,d0
int8x16_t  vnegq_s8(int8x16_t a);    // VNEG.S8 q0,q0
int16x8_t  vnegq_s16(int16x8_t a);   // VNEG.S16 q0,q0
int32x4_t  vnegq_s32(int32x4_t a);   // VNEG.S32 q0,q0
float32x4_t vnegq_f32(float32x4_t a); // VNEG.F32 q0,q0

```

Saturating Negate: $\text{sat}(Vd[i] = -Va[i])$

```

int8x8_t   vqneg_s8(int8x8_t a);      // VQNEG.S8 d0,d0
int16x4_t  vqneg_s16(int16x4_t a);    // VQNEG.S16 d0,d0
int32x2_t  vqneg_s32(int32x2_t a);    // VQNEG.S32 d0,d0
int8x16_t  vqnegq_s8(int8x16_t a);    // VQNEG.S8 q0,q0
int16x8_t  vqnegq_s16(int16x8_t a);   // VQNEG.S16 q0,q0
int32x4_t  vqnegq_s32(int32x4_t a);   // VQNEG.S32 q0,q0

```

Count leading sign bits

```

int8x8_t   vcsls_s8(int8x8_t a);      // VCLS.S8 d0,d0
int16x4_t  vcsls_s16(int16x4_t a);    // VCLS.S16 d0,d0
int32x2_t  vcsls_s32(int32x2_t a);    // VCLS.S32 d0,d0
int8x16_t  vcslsq_s8(int8x16_t a);    // VCLS.S8 q0,q0
int16x8_t  vcslsq_s16(int16x8_t a);   // VCLS.S16 q0,q0
int32x4_t  vcslsq_s32(int32x4_t a);   // VCLS.S32 q0,q0

```

Count leading zeros

```

int8x8_t   vclz_s8(int8x8_t a);      // VCLZ.I8 d0,d0
int16x4_t  vclz_s16(int16x4_t a);    // VCLZ.I16 d0,d0
int32x2_t  vclz_s32(int32x2_t a);    // VCLZ.I32 d0,d0
uint8x8_t  vclz_u8(uint8x8_t a);     // VCLZ.I8 d0,d0
uint16x4_t vclz_u16(uint16x4_t a);   // VCLZ.I16 d0,d0
uint32x2_t vclz_u32(uint32x2_t a);   // VCLZ.I32 d0,d0
int8x16_t  vclzq_s8(int8x16_t a);    // VCLZ.I8 q0,q0
int16x8_t  vclzq_s16(int16x8_t a);   // VCLZ.I16 q0,q0
int32x4_t  vclzq_s32(int32x4_t a);   // VCLZ.I32 q0,q0
uint8x16_t vclzq_u8(uint8x16_t a);   // VCLZ.I8 q0,q0
uint16x8_t vclzq_u16(uint16x8_t a);  // VCLZ.I16 q0,q0
uint32x4_t vclzq_u32(uint32x4_t a);  // VCLZ.I32 q0,q0

```

Count number of set bits

```

uint8x8_t  vcnt_u8(uint8x8_t a);     // VCNT.8 d0,d0
int8x8_t   vcnt_s8(int8x8_t a);      // VCNT.8 d0,d0
poly8x8_t  vcnt_p8(poly8x8_t a);     // VCNT.8 d0,d0
uint8x16_t vcntq_u8(uint8x16_t a);   // VCNT.8 q0,q0
int8x16_t  vcntq_s8(int8x16_t a);    // VCNT.8 q0,q0
poly8x16_t vcntq_p8(poly8x16_t a);   // VCNT.8 q0,q0

```

Reciprocal estimate

```

float32x2_t vrecpe_f32(float32x2_t a); // VRECPE.F32 d0,d0
uint32x2_t  vrecpe_u32(uint32x2_t a);  // VRECPE.U32 d0,d0
float32x4_t vrecpeq_f32(float32x4_t a); // VRECPE.F32 q0,q0
uint32x4_t  vrecpeq_u32(uint32x4_t a);  // VRECPE.U32 q0,q0

```

Reciprocal square root estimate

```

float32x2_t vrsqrte_f32(float32x2_t a); // VRSQRTE.F32 d0,d0
uint32x2_t  vrsqrte_u32(uint32x2_t a);  // VRSQRTE.U32 d0,d0
float32x4_t vrsqrteq_f32(float32x4_t a); // VRSQRTE.F32 q0,q0
uint32x4_t  vrsqrteq_u32(uint32x4_t a);  // VRSQRTE.U32 q0,q0

```

E.3.29 Logical operations

These intrinsics provide bitwise logical operations.

Bitwise not

```

int8x8_t   vmvn_s8(int8x8_t a);      // VMVN d0,d0
int16x4_t  vmvn_s16(int16x4_t a);    // VMVN d0,d0
int32x2_t  vmvn_s32(int32x2_t a);    // VMVN d0,d0

```

```

uint8x8_t  vmvn_u8(uint8x8_t a);      // VMVN d0,d0
uint16x4_t vmvn_u16(uint16x4_t a);    // VMVN d0,d0
uint32x2_t vmvn_u32(uint32x2_t a);    // VMVN d0,d0
poly8x8_t  vmvn_p8(poly8x8_t a);      // VMVN d0,d0
int8x16_t  vmvnq_s8(int8x16_t a);     // VMVN q0,q0
int16x8_t  vmvnq_s16(int16x8_t a);    // VMVN q0,q0
int32x4_t  vmvnq_s32(int32x4_t a);    // VMVN q0,q0
uint8x16_t vmvnq_u8(uint8x16_t a);    // VMVN q0,q0
uint16x8_t vmvnq_u16(uint16x8_t a);   // VMVN q0,q0
uint32x4_t vmvnq_u32(uint32x4_t a);   // VMVN q0,q0
poly8x16_t vmvnq_p8(poly8x16_t a);    // VMVN q0,q0

```

Bitwise and

```

int8x8_t  vand_s8(int8x8_t a, int8x8_t b);      // VAND d0,d0,d0
int16x4_t vand_s16(int16x4_t a, int16x4_t b);   // VAND d0,d0,d0
int32x2_t vand_s32(int32x2_t a, int32x2_t b);   // VAND d0,d0,d0
int64x1_t vand_s64(int64x1_t a, int64x1_t b);   // VAND d0,d0,d0
uint8x8_t vand_u8(uint8x8_t a, uint8x8_t b);    // VAND d0,d0,d0
uint16x4_t vand_u16(uint16x4_t a, uint16x4_t b); // VAND d0,d0,d0
uint32x2_t vand_u32(uint32x2_t a, uint32x2_t b); // VAND d0,d0,d0
uint64x1_t vand_u64(uint64x1_t a, uint64x1_t b); // VAND d0,d0,d0
int8x16_t vandq_s8(int8x16_t a, int8x16_t b);   // VAND q0,q0,q0
int16x8_t vandq_s16(int16x8_t a, int16x8_t b);  // VAND q0,q0,q0
int32x4_t vandq_s32(int32x4_t a, int32x4_t b);  // VAND q0,q0,q0
int64x2_t vandq_s64(int64x2_t a, int64x2_t b);  // VAND q0,q0,q0
uint8x16_t vandq_u8(uint8x16_t a, uint8x16_t b); // VAND q0,q0,q0
uint16x8_t vandq_u16(uint16x8_t a, uint16x8_t b); // VAND q0,q0,q0
uint32x4_t vandq_u32(uint32x4_t a, uint32x4_t b); // VAND q0,q0,q0
uint64x2_t vandq_u64(uint64x2_t a, uint64x2_t b); // VAND q0,q0,q0

```

Bitwise or

```

int8x8_t  vorr_s8(int8x8_t a, int8x8_t b);      // VORR d0,d0,d0
int16x4_t vorr_s16(int16x4_t a, int16x4_t b);   // VORR d0,d0,d0
int32x2_t vorr_s32(int32x2_t a, int32x2_t b);   // VORR d0,d0,d0
int64x1_t vorr_s64(int64x1_t a, int64x1_t b);   // VORR d0,d0,d0
uint8x8_t vorr_u8(uint8x8_t a, uint8x8_t b);    // VORR d0,d0,d0
uint16x4_t vorr_u16(uint16x4_t a, uint16x4_t b); // VORR d0,d0,d0
uint32x2_t vorr_u32(uint32x2_t a, uint32x2_t b); // VORR d0,d0,d0
uint64x1_t vorr_u64(uint64x1_t a, uint64x1_t b); // VORR d0,d0,d0
int8x16_t vorrq_s8(int8x16_t a, int8x16_t b);   // VORR q0,q0,q0
int16x8_t vorrq_s16(int16x8_t a, int16x8_t b);  // VORR q0,q0,q0
int32x4_t vorrq_s32(int32x4_t a, int32x4_t b);  // VORR q0,q0,q0
int64x2_t vorrq_s64(int64x2_t a, int64x2_t b);  // VORR q0,q0,q0
uint8x16_t vorrq_u8(uint8x16_t a, uint8x16_t b); // VORR q0,q0,q0
uint16x8_t vorrq_u16(uint16x8_t a, uint16x8_t b); // VORR q0,q0,q0
uint32x4_t vorrq_u32(uint32x4_t a, uint32x4_t b); // VORR q0,q0,q0
uint64x2_t vorrq_u64(uint64x2_t a, uint64x2_t b); // VORR q0,q0,q0

```

Bitwise exclusive or (EOR or XOR)

```

int8x8_t   veor_s8(int8x8_t a, int8x8_t b);           // VEOR d0,d0,d0
int16x4_t  veor_s16(int16x4_t a, int16x4_t b);        // VEOR d0,d0,d0
int32x2_t  veor_s32(int32x2_t a, int32x2_t b);        // VEOR d0,d0,d0
int64x1_t  veor_s64(int64x1_t a, int64x1_t b);        // VEOR d0,d0,d0
uint8x8_t  veor_u8(uint8x8_t a, uint8x8_t b);         // VEOR d0,d0,d0
uint16x4_t veor_u16(uint16x4_t a, uint16x4_t b);      // VEOR d0,d0,d0
uint32x2_t veor_u32(uint32x2_t a, uint32x2_t b);      // VEOR d0,d0,d0
uint64x1_t veor_u64(uint64x1_t a, uint64x1_t b);      // VEOR d0,d0,d0
int8x16_t  veorq_s8(int8x16_t a, int8x16_t b);        // VEOR q0,q0,q0
int16x8_t  veorq_s16(int16x8_t a, int16x8_t b);       // VEOR q0,q0,q0
int32x4_t  veorq_s32(int32x4_t a, int32x4_t b);       // VEOR q0,q0,q0
int64x2_t  veorq_s64(int64x2_t a, int64x2_t b);       // VEOR q0,q0,q0
uint8x16_t veorq_u8(uint8x16_t a, uint8x16_t b);      // VEOR q0,q0,q0
uint16x8_t veorq_u16(uint16x8_t a, uint16x8_t b);     // VEOR q0,q0,q0
uint32x4_t veorq_u32(uint32x4_t a, uint32x4_t b);     // VEOR q0,q0,q0
uint64x2_t veorq_u64(uint64x2_t a, uint64x2_t b);     // VEOR q0,q0,q0

```

Bit Clear

```

int8x8_t   vbic_s8(int8x8_t a, int8x8_t b);           // VBIC d0,d0,d0
int16x4_t  vbic_s16(int16x4_t a, int16x4_t b);        // VBIC d0,d0,d0
int32x2_t  vbic_s32(int32x2_t a, int32x2_t b);        // VBIC d0,d0,d0
int64x1_t  vbic_s64(int64x1_t a, int64x1_t b);        // VBIC d0,d0,d0
uint8x8_t  vbic_u8(uint8x8_t a, uint8x8_t b);         // VBIC d0,d0,d0
uint16x4_t vbic_u16(uint16x4_t a, uint16x4_t b);      // VBIC d0,d0,d0
uint32x2_t vbic_u32(uint32x2_t a, uint32x2_t b);      // VBIC d0,d0,d0
uint64x1_t vbic_u64(uint64x1_t a, uint64x1_t b);      // VBIC d0,d0,d0
int8x16_t  vbicq_s8(int8x16_t a, int8x16_t b);        // VBIC q0,q0,q0
int16x8_t  vbicq_s16(int16x8_t a, int16x8_t b);       // VBIC q0,q0,q0
int32x4_t  vbicq_s32(int32x4_t a, int32x4_t b);       // VBIC q0,q0,q0
int64x2_t  vbicq_s64(int64x2_t a, int64x2_t b);       // VBIC q0,q0,q0
uint8x16_t vbicq_u8(uint8x16_t a, uint8x16_t b);      // VBIC q0,q0,q0
uint16x8_t vbicq_u16(uint16x8_t a, uint16x8_t b);     // VBIC q0,q0,q0
uint32x4_t vbicq_u32(uint32x4_t a, uint32x4_t b);     // VBIC q0,q0,q0
uint64x2_t vbicq_u64(uint64x2_t a, uint64x2_t b);     // VBIC q0,q0,q0

```

Bitwise OR complement

```

int8x8_t   vorn_s8(int8x8_t a, int8x8_t b);           // VORN d0,d0,d0
int16x4_t  vorn_s16(int16x4_t a, int16x4_t b);        // VORN d0,d0,d0
int32x2_t  vorn_s32(int32x2_t a, int32x2_t b);        // VORN d0,d0,d0
int64x1_t  vorn_s64(int64x1_t a, int64x1_t b);        // VORN d0,d0,d0
uint8x8_t  vorn_u8(uint8x8_t a, uint8x8_t b);         // VORN d0,d0,d0
uint16x4_t vorn_u16(uint16x4_t a, uint16x4_t b);      // VORN d0,d0,d0
uint32x2_t vorn_u32(uint32x2_t a, uint32x2_t b);      // VORN d0,d0,d0
uint64x1_t vorn_u64(uint64x1_t a, uint64x1_t b);      // VORN d0,d0,d0
int8x16_t  vornq_s8(int8x16_t a, int8x16_t b);        // VORN q0,q0,q0

```

```

int16x8_t vorrq_s16(int16x8_t a, int16x8_t b);    // VORN q0,q0,q0
int32x4_t vorrq_s32(int32x4_t a, int32x4_t b);    // VORN q0,q0,q0
int64x2_t vorrq_s64(int64x2_t a, int64x2_t b);    // VORN q0,q0,q0
uint8x16_t vorrq_u8(uint8x16_t a, uint8x16_t b);  // VORN q0,q0,q0
uint16x8_t vorrq_u16(uint16x8_t a, uint16x8_t b); // VORN q0,q0,q0
uint32x4_t vorrq_u32(uint32x4_t a, uint32x4_t b); // VORN q0,q0,q0
uint64x2_t vorrq_u64(uint64x2_t a, uint64x2_t b); // VORN q0,q0,q0

```

Bitwise Select

Note

This intrinsic can compile to any of VBSL/VBIF/VBIT depending on register allocation.

```

int8x8_t  vbsl_s8(uint8x8_t a, int8x8_t b, int8x8_t c);    // VBSL d0,d0,d0
int16x4_t vbsl_s16(uint16x4_t a, int16x4_t b, int16x4_t c); // VBSL d0,d0,d0
int32x2_t vbsl_s32(uint32x2_t a, int32x2_t b, int32x2_t c); // VBSL d0,d0,d0
int64x1_t vbsl_s64(uint64x1_t a, int64x1_t b, int64x1_t c); // VBSL d0,d0,d0
uint8x8_t vbsl_u8(uint8x8_t a, uint8x8_t b, uint8x8_t c);   // VBSL d0,d0,d0
uint16x4_t vbsl_u16(uint16x4_t a, uint16x4_t b, uint16x4_t c); // VBSL d0,d0,d0
uint32x2_t vbsl_u32(uint32x2_t a, uint32x2_t b, uint32x2_t c); // VBSL d0,d0,d0
uint64x1_t vbsl_u64(uint64x1_t a, uint64x1_t b, uint64x1_t c); // VBSL d0,d0,d0
float32x2_t vbsl_f32(uint32x2_t a, float32x2_t b, float32x2_t c); // VBSL d0,d0,d0
poly8x8_t vbsl_p8(uint8x8_t a, poly8x8_t b, poly8x8_t c);   // VBSL d0,d0,d0
poly16x4_t vbsl_p16(uint16x4_t a, poly16x4_t b, poly16x4_t c); // VBSL d0,d0,d0
int8x16_t vbslq_s8(uint8x16_t a, int8x16_t b, int8x16_t c); // VBSL q0,q0,q0
int16x8_t vbslq_s16(uint16x8_t a, int16x8_t b, int16x8_t c); // VBSL q0,q0,q0
int32x4_t vbslq_s32(uint32x4_t a, int32x4_t b, int32x4_t c); // VBSL q0,q0,q0
int64x2_t vbslq_s64(uint64x2_t a, int64x2_t b, int64x2_t c); // VBSL q0,q0,q0
uint8x16_t vbslq_u8(uint8x16_t a, uint8x16_t b, uint8x16_t c); // VBSL q0,q0,q0
uint16x8_t vbslq_u16(uint16x8_t a, uint16x8_t b, uint16x8_t c); // VBSL q0,q0,q0
uint32x4_t vbslq_u32(uint32x4_t a, uint32x4_t b, uint32x4_t c); // VBSL q0,q0,q0
uint64x2_t vbslq_u64(uint64x2_t a, uint64x2_t b, uint64x2_t c); // VBSL q0,q0,q0
float32x4_t vbslq_f32(uint32x4_t a, float32x4_t b, float32x4_t c); // VBSL q0,q0,q0
poly8x16_t vbslq_p8(uint8x16_t a, poly8x16_t b, poly8x16_t c); // VBSL q0,q0,q0
poly16x8_t vbslq_p16(uint16x8_t a, poly16x8_t b, poly16x8_t c); // VBSL q0,q0,q0

```

E.3.30 Transposition operations

This intrinsics provide transposition operations.

Transpose elements

```

int8x8x2_t vtrn_s8(int8x8_t a, int8x8_t b);    // VTRN.8 d0,d0
int16x4x2_t vtrn_s16(int16x4_t a, int16x4_t b); // VTRN.16 d0,d0
int32x2x2_t vtrn_s32(int32x2_t a, int32x2_t b); // VTRN.32 d0,d0
uint8x8x2_t vtrn_u8(uint8x8_t a, uint8x8_t b);  // VTRN.8 d0,d0
uint16x4x2_t vtrn_u16(uint16x4_t a, uint16x4_t b); // VTRN.16 d0,d0

```

```

uint32x2x2_t vtrn_u32(uint32x2_t a, uint32x2_t b); // VTRN.32 d0,d0
float32x2x2_t vtrn_f32(float32x2_t a, float32x2_t b); // VTRN.32 d0,d0
poly8x8x2_t vtrn_p8(poly8x8_t a, poly8x8_t b); // VTRN.8 d0,d0
poly16x4x2_t vtrn_p16(poly16x4_t a, poly16x4_t b); // VTRN.16 d0,d0
int8x16x2_t vtrnq_s8(int8x16_t a, int8x16_t b); // VTRN.8 q0,q0
int16x8x2_t vtrnq_s16(int16x8_t a, int16x8_t b); // VTRN.16 q0,q0
int32x4x2_t vtrnq_s32(int32x4_t a, int32x4_t b); // VTRN.32 q0,q0
uint8x16x2_t vtrnq_u8(uint8x16_t a, uint8x16_t b); // VTRN.8 q0,q0
uint16x8x2_t vtrnq_u16(uint16x8_t a, uint16x8_t b); // VTRN.16 q0,q0
uint32x4x2_t vtrnq_u32(uint32x4_t a, uint32x4_t b); // VTRN.32 q0,q0
float32x4x2_t vtrnq_f32(float32x4_t a, float32x4_t b); // VTRN.32 q0,q0
poly8x16x2_t vtrnq_p8(poly8x16_t a, poly8x16_t b); // VTRN.8 q0,q0
poly16x8x2_t vtrnq_p16(poly16x8_t a, poly16x8_t b); // VTRN.16 q0,q0

```

Interleave elements

```

int8x8x2_t vzip_s8(int8x8_t a, int8x8_t b); // VZIP.8 d0,d0
int16x4x2_t vzip_s16(int16x4_t a, int16x4_t b); // VZIP.16 d0,d0
uint8x8x2_t vzip_u8(uint8x8_t a, uint8x8_t b); // VZIP.8 d0,d0
uint16x4x2_t vzip_u16(uint16x4_t a, uint16x4_t b); // VZIP.16 d0,d0
float32x2x2_t vzip_f32(float32x2_t a, float32x2_t b); // VZIP.32 d0,d0
poly8x8x2_t vzip_p8(poly8x8_t a, poly8x8_t b); // VZIP.8 d0,d0
poly16x4x2_t vzip_p16(poly16x4_t a, poly16x4_t b); // VZIP.16 d0,d0
int8x16x2_t vzipq_s8(int8x16_t a, int8x16_t b); // VZIP.8 q0,q0
int16x8x2_t vzipq_s16(int16x8_t a, int16x8_t b); // VZIP.16 q0,q0
int32x4x2_t vzipq_s32(int32x4_t a, int32x4_t b); // VZIP.32 q0,q0
uint8x16x2_t vzipq_u8(uint8x16_t a, uint8x16_t b); // VZIP.8 q0,q0
uint16x8x2_t vzipq_u16(uint16x8_t a, uint16x8_t b); // VZIP.16 q0,q0
uint32x4x2_t vzipq_u32(uint32x4_t a, uint32x4_t b); // VZIP.32 q0,q0
float32x4x2_t vzipq_f32(float32x4_t a, float32x4_t b); // VZIP.32 q0,q0
poly8x16x2_t vzipq_p8(poly8x16_t a, poly8x16_t b); // VZIP.8 q0,q0
poly16x8x2_t vzipq_p16(poly16x8_t a, poly16x8_t b); // VZIP.16 q0,q0

```

De-Interleave elements

```

int8x8x2_t vuzp_s8(int8x8_t a, int8x8_t b); // VUZP.8 d0,d0
int16x4x2_t vuzp_s16(int16x4_t a, int16x4_t b); // VUZP.16 d0,d0
int32x2x2_t vuzp_s32(int32x2_t a, int32x2_t b); // VUZP.32 d0,d0
uint8x8x2_t vuzp_u8(uint8x8_t a, uint8x8_t b); // VUZP.8 d0,d0
uint16x4x2_t vuzp_u16(uint16x4_t a, uint16x4_t b); // VUZP.16 d0,d0
uint32x2x2_t vuzp_u32(uint32x2_t a, uint32x2_t b); // VUZP.32 d0,d0
float32x2x2_t vuzp_f32(float32x2_t a, float32x2_t b); // VUZP.32 d0,d0
poly8x8x2_t vuzp_p8(poly8x8_t a, poly8x8_t b); // VUZP.8 d0,d0
poly16x4x2_t vuzp_p16(poly16x4_t a, poly16x4_t b); // VUZP.16 d0,d0
int8x16x2_t vuzpq_s8(int8x16_t a, int8x16_t b); // VUZP.8 q0,q0
int16x8x2_t vuzpq_s16(int16x8_t a, int16x8_t b); // VUZP.16 q0,q0
int32x4x2_t vuzpq_s32(int32x4_t a, int32x4_t b); // VUZP.32 q0,q0
uint8x16x2_t vuzpq_u8(uint8x16_t a, uint8x16_t b); // VUZP.8 q0,q0
uint16x8x2_t vuzpq_u16(uint16x8_t a, uint16x8_t b); // VUZP.16 q0,q0

```

```
uint32x4x2_t vuzpq_u32(uint32x4_t a, uint32x4_t b); // VUZP.32 q0,q0
float32x4x2_t vuzpq_f32(float32x4_t a, float32x4_t b); // VUZP.32 q0,q0
poly8x16x2_t vuzpq_p8(poly8x16_t a, poly8x16_t b); // VUZP.8 q0,q0
poly16x8x2_t vuzpq_p16(poly16x8_t a, poly16x8_t b); // VUZP.16 q0,q0
```

E.3.31 Vector reinterpret cast operations

In some situations, you might want to treat a vector as having a different type, without changing its value. A set of intrinsics is provided to perform this type of conversion.

Syntax

```
vreinterpret{q}_dsttype_srctype
```

Where:

<i>q</i>	Specifies that the conversion operates on 128-bit vectors. If it is not present, the conversion operates on 64-bit vectors.
<i>dsttype</i>	Represents the type to convert to.
<i>srctype</i>	Represents the type being converted.

Example

The following intrinsic reinterprets a vector of four signed 16-bit integers as a vector of four unsigned integers:

```
uint16x4_t vreinterpret_u16_s16(int16x4_t a);
```

The following intrinsic reinterprets a vector of four 32-bit floating point values integers as a vector of four signed integers.

```
int8x16_t vreinterpretq_s8_f32(float32x4_t a);
```

These conversions do not change the bit pattern represented by the vector.

