RealView® 编译工具

4.0 版

编译器参考指南

ARM®

ARM DUI 0348BC (ID090708)

RealView 编译工具
编译器参考指南

版本信息

本手册进行了以下更改。

更改历史记录

日期	发行号	保密性	变更
2007 年 3 月	A	非保密	RealView Development Suite v3.1 3.1 版
2008 年 9 月	B	非保密	RealView Development Suite v4.0 4.0 版
2009 年 1 月 23 日	B	非保密	RealView Development Suite 4.0 版的文档更新
2009 年 3 月 2 日	B	非保密	RealView Development Suite 4.0 版的文档更新 2

所有权声明

保密状态

本文档的内容是非保密的。根据 ARM 与 ARM 将本文档交予的参与方的协议条款，使用、复制和公开本文档内容的权利可能会受到许可限制的制约。

受限访问是一种 ARM 内部分类。

产品状态

本文档的信息是开发的产品的最新信息。

网址

`http://www.arm.com`

目录
RealView 编译工具
编译器参考指南

前言

本前言介绍了 *《RealView 编译工具编译器参考指南》*。它分为以下几节：

- 第 vi 页的*关于本手册*
- 第 x 页的*反馈*

关于本手册

本手册提供有关 *RealView 编译工具* (RVCT) 的参考信息，并介绍了 ARM 编译器的命令行选项。本手册也提供有关 ARM 如何在编译器中实现 C 和 C++ 的参考材料。有关使用和控制 ARM 编译器的一般信息，请参阅 《RVCT 编译器用户指南》。

适用对象

本手册是为所有使用 RVCT 编写应用程序的开发者编写的。本手册假定您是一位经验丰富的软件开发人员。有关 RVCT 附带的 ARM 开发工具的概述，请参阅 《RealView 编译工具要点指南》。

使用本手册

本手册由以下章节和附录组成：

第 1 章 *简介*

阅读本章后，可以大致了解 ARM 编译器、标准一致性以及 C 和 C++ 库。

第 2 章 *编译器命令行选项*

本章列出了 ARM 编译器接受的所有命令行选项。

第 3 章 *语言扩展*

本章介绍了 ARM 编译器提供的语言扩展，并提供了标准遵从性和实现方法的详细信息。

第 4 章 *编译器特有的功能*

本章详细列出了 ARM 特有的关键字、运算符、编译指示、内在函数、宏和半主机*超级用户调用* (SVC)。

第 5 章 *C 和 C++ 实现细节*

本章介绍了 ARM 编译器的语言实现详细信息。

附录 A *via 文件语法*

本附录介绍了 via 文件的语法。您可以使用 via 文件为很多 ARM 工具指定命令行参数。

附录 B *标准 C 实现定义*

本附录提供有关 ARM C 实现的信息，该实现直接与 ISO C 要求相关。

附录 C *标准 C++ 实现定义*

本附录提供了有关 ARM C++ 实现的信息。

附录 D *C 和 C++ 编译器实现限制*

本附录提供了有关 ARM 编译器中 C 和 C++ 实现的限制。

附录 E *使用 NEON 支持*

本附录介绍了此版本的 RVCT 所支持的 NEON™ 内在函数的相关信息。

本手册假定 ARM 软件安装在缺省位置。例如，在 Windows 上，这可能是 *volume*:\Program Files\ARM。引用路径名时，假定安装位置为 *install_directory*，如 *install_directory*\Documentation\...。如果将 ARM 软件安装在其他位置，则可能需要更改此位置。

印刷约定

本手册使用以下印刷约定：

`monospace` 表示可以从键盘输入的文本，如命令、文件和程序名以及源代码。

`mono`space 表示允许的命令或选项缩写。可只输入下划线标记的文本，无需输入命令或选项的全名。

`monospace italic`

表示此处的命令和函数的变量可用特定值代替。

等宽粗体 表示在示例代码以外使用的语言关键字。

斜体 突出显示重要注释、介绍特殊术语以及表示内部交叉引用和引文。

粗体 突出显示界面元素，如菜单名称。有时候也用在描述性列表中以示强调，以及表示 ARM 处理器信号名称。

更多参考出版物

本部分列出了 ARM 公司和第三方发布的、可提供有关 ARM 系列处理器开发代码的附加信息的出版物。

ARM 公司将定期对其文档进行更新和更正。有关最新勘误表、附录和 ARM *常见问题* (FAQ)，请访问 http://infocenter.arm.com/help/index.jsp。

ARM 公司出版物

本手册包含的参考信息专用于随 RVCT 提供的开发工具。该套件中包含的其他出版物有：

- 《RVCT 要点指南》 (ARM DUI 0202)
- *《RVCT 编译器用户指南》* (ARM DUI 0205)
- 《RVCT 库和浮点支持指南》 (ARM DUI 0349)
- 《RVCT 链接器用户指南》 (ARM DUI 0206)
- 《RVCT 链接器参考指南》 (ARM DUI 0381)
- 《RVCT 实用程序指南》 (ARM DUI 0382)
- 《RVCT 汇编器指南》 (ARM DUI 0204)
- 《RVCT 开发指南》 (ARM DUI 0203)

有关基本标准、软件接口和 ARM 支持的标准的完整信息，请参阅 *install_directory*\Documentation\Specifications\...。

此外，有关与 ARM 产品相关的特定信息，请参阅下列文档：

- 《ARM 体系结构参考手册》， ARMv7-A 和 ARMv7-R 版 (ARM DDI 0406)
- 《ARM7-M 体系结构参考手册》 (ARM DDI 0403)
- 《ARM6-M 体系结构参考手册》 (ARM DDI 0419)
- 您的硬件设备的 ARM 数据手册或技术参考手册。

其他出版物

以下出版物提供了有关 ETSI 基本运算的信息。要获取这些信息，请访问*国际电信联盟* (ITU) 无线电通信局的网站 http://www.itu.int。

- ETSI 建议 G.191：《语音和音频编码标准化软件工具》 (*Software tools for speech and audio coding standardization*)
- *《ITU-T 软件工具库 2005 用户手册》*，收录为 ETSI 建议书 G.191 的一部分
- ETSI 建议 G723.1：《传输速率为 5.3 和 6.3 Kb/s 的多媒体通信双速率语音编码器》 (*Dual rate speech coder for multimedia communications transmitting at 5.3 and 6.3 kbit/s*)
- ETSI 建议 G.729：《使用共轭结构代数码激励线性预测 (CS-ACELP) 的 8 Kb/s 语音编码》 (*Coding of speech at 8 kbit/s using conjugate-structure algebraic-code-excited linear prediction (CS-ACELP)*)。

德州仪器公司的网站 http://www.ti.com 上提供有关 TI 编译器内在函数信息的出版物。

反馈

ARM Limited 欢迎提供有关 RealView 编译工具与文档的反馈。

对 RealView 编译工具的反馈

如果您有关于 RVCT 的任何问题，请与您的供应商联系。为便于供应商快速提供有用的答复，请提供：

- 您的姓名和公司
- 产品序列号
- 您所用版本的详细信息
- 您运行的平台的详细信息，如硬件平台、操作系统类型和版本
- 能重现问题的一小段独立的程序
- 您预期发生和实际发生的情况的详细说明
- 您使用的命令，包括所有命令行选项
- 能说明问题的示例输出
- 工具的版本字符串，包括版本号和内部版本号。

关于本手册的反馈

如果您发现本手册有任何错误或遗漏之处，请发送电子邮件到 errata@arm.com，并提供：

- 文档标题
- 文档编号
- 您有疑问的页码
- 问题的简要说明

我们还欢迎您对需要增加和改进之处提出建议。

第 1 章
简介

本章介绍随 *RealView 编译工具* (RVCT) 一起提供的 ARM 编译器。它将介绍遵循的标准，并概述 RVCT 附带的运行时库。本附录分为以下几节：

- 第 1-2 页的*关于 ARM 编译器*
- 第 1-3 页的*源语言模式*
- 第 1-5 页的*语言扩展和语言遵从性*
- 第 1-7 页的*C 和 C++ 库*

1.1 关于 ARM 编译器

利用 ARM 编译器 armcc 可以编译 C 和 C++ 代码。

编译器：

- 是一种成熟的编译器。利用命令行选项，可以控制优化级别。
- 可以将：
  - ISO 标准 C:1990 源代码
  - ISO 标准 C:1999 源代码
  - ISO 标准 C++:2003 源代码

  编译为：
  - 32 位 ARM 代码
  - 16/32 位 Thumb-2 代码
  - 16 位 Thumb 代码。
- 遵循 *ARM 体系结构的基础标准应用程序二进制接口* (BSABI)。特别是，编译器可以：
  - 生成 ELF 格式的输出对象。
  - 生成 DWARF 调试标准版本 3 (DWARF 3) 调试信息。 RVCT 还支持 DWARF 2 调试表。

  有关详细信息，请参阅 *《库和浮点支持指南》*中第1-3 页的*ARM 体系结构 ABI 遵从性*。
- 可以生成输出代码的汇编语言列表，并且可以交叉存取源代码的汇编语言列表。

如果是从先前的版本升级到 RVCT 或者是初次使用 RVCT，请务必阅读《RealView 编译工具要点指南》以了解最新信息。

1.2 源语言模式

ARM 编译器有三种不同的源语言模式可用于编译不同种类的 C 和 C++ 源代码。这些模式是：

- ISO C90
- ISO C99
- ISO C++。

1.2.1 ISO C90

ARM 编译器根据 1990 C 标准和附录的定义编译 C 代码：

- ISO/IEC 9899:1990。 1990 C 国际标准。
- ISO/IEC 9899 AM1。 1995 标准附录 1，通过 `wchar.h` 和 `wtype.h` 增加了对国际字符的支持。

ARM 编译器还支持几种 ISO C90 扩展。有关详细信息，请参阅第1-5 页的*语言扩展和语言遵从性*。

在整篇文档中，术语：

C90 是指 ISO C90，以及 ARM 扩展。

使用编译器选项 `--c90` 可编译 C90 代码。这是缺省设置。

严格 C90 是指根据 1990 C 标准和附录定义的 C。

另请参阅

- 第2-22 页的*`--c90`*
- 第1-5 页的*语言扩展和语言遵从性*
- 附录 B *标准 C 实现定义*

1.2.2 ISO C99

ARM 编译器编译根据 1999 C 标准和附录定义的 C 代码：

- ISO/IEC 9899:1999。 1999 C 国际标准。

ARM 编译器还支持几种 ISO C99 扩展。有关详细信息，请参阅第1-5 页的*语言扩展和语言遵从性*。

在整篇文档中，术语：

C99 是指 ISO C99，以及 ARM 和 GNU 扩展。
要编译 C99 代码，可使用编译器选项 `--c99`。

严格 C99 是指根据 1999 C 标准和附录定义的 C。

标准 C 是指 C90 或 C99 （视具体情况）。

C 是指 C90、严格 C90、 C99 和标准 C 中的任何一种。

另请参阅
- 第2-22 页的*--c99*
- 第1-5 页的*语言扩展和语言遵从性*
- 附录 B *标准 C 实现定义*

1.2.3 ISO C++

ARM 编译器编译根据 2003 标准而定义的 C++，宽流和导出模板除外：

- ISO/IEC 14822:2003。 2003 C++ 国际标准。

ARM 编译器还支持几种 ISO C++ 扩展。有关详细信息，请参阅第1-5 页的*语言扩展和语言遵从性*。

在整篇文档中，术语：

严格 C++ 是指 ISO C++，宽流和导出模板除外。

标准 C++ 是指严格 C++。

C++ 是指 ISO C++ （宽流和导出模板除外），含有或不含 ARM 扩展。
要编译 C++ 代码，可使用编译器选项 `--cpp`。

另请参阅
- 第2-30 页的*--cpp*
- 第1-5 页的*语言扩展和语言遵从性*
- 附录 C *标准 C++ 实现定义*.

1.3 语言扩展和语言遵从性

编译器支持大量对各种源语言的扩展。它还提供几个命令行选项，以控制与可用源语言的遵从性。

1.3.1 语言扩展

编译器支持的语言扩展按如下分类：

C99 功能 编译器可以使用 C99 的某些语言功能：

- 严格 C90 的扩展，如 // 样式的注释
- 标准 C++ 和严格 C90 的扩展，如 `restrict` 指针。

有关详细信息，请参阅：

- 第3-4 页的*C90 中提供的 C99 语言功能*
- 第3-6 页的*C++ 和 C90 中提供的 C99 语言功能*

标准 C 扩展 编译器支持多个严格 C99 扩展，例如，重写旧式非原型定义的函数原型。有关详细信息，请参阅第3-9 页的*标准 C 语言扩展*。

C90 中也提供了这些标准 C 扩展。

标准 C++ 扩展 编译器支持多个严格 C++ 扩展，例如，类成员声明中的限定名称。有关详细信息，请参阅第3-14 页的*标准 C++ 语言扩展*。

标准 C 和 C90 中都不提供这些扩展。

标准 C 和标准 C++ 扩展

编译器支持严格 C++ 和严格 C90 特有的某些扩展，如匿名类、结构和联合。有关详细信息，请参阅第3-18 页的*标准 C 和标准 C++ 语言扩展*。

GNU 扩展 编译器支持 GNU 编译器提供的某些扩展，例如 GNU 样式的扩展左值和 GNU 内置函数。有关详细信息，请参阅：

- 第1-6 页的*语言遵从性*
- 第3-23 页的*GNU 语言扩展*
- 第 4 章 *编译器特有的功能*

ARM 特有的扩展 编译器支持 ARM 编译器特有的一系列扩展，例如指令内在函数和其他内置函数。有关详细信息，请参阅第 4 章 *编译器特有的功能*。

1.3.2 语言遵从性

编译器有几种模式，在这些模式中，有的模式要求必须遵从源语言，有的模式则不做硬性规定：

Strict 模式 在 strict 模式下，编译器强制与源语言的语言标准保持一致。例如，编译严格 C90 时若使用 // 样式的注释，将导致错误。

若要在 strict 模式下进行编译，请使用命令行选项 `--strict`。

GNU 模式 在 GNU 模式下，相关源语言的所有 GNU 编译器扩展都可用。例如，在 GNU 模式中：

- 源语言是 C90、C99 或非严格 C++ 中的任何一种时，可以在 **switch** 语句中使用条件范围
- 源语言是 C90 或非严格 C++ 时，可以使用 C99 样式的指定初始值设定项

若要在 GNU 模式下进行编译，请使用编译器选项 `--gnu`。

注意

在非严格模式下时还可以使用某些 GNU 扩展。

示例

以下示例说明如何将源语言模式与语言遵从模式结合使用：

- 使用命令行选项 `--strict` 编译 `.cpp` 文件时将编译标准 C++
- 使用命令行选项 `--gnu` 编译 C 源文件将编译 GNU 模式 C90
- 使用命令行选项 `--strict` 和 `--gnu` 编译 `.c` 文件会出错。

另请参阅

- 第2-66 页的*--gnu*
- 第2-115 页的*--strict*, *--no_strict*
- 第3-23 页的*GNU 语言扩展*
- *《编译器用户指南》*中第2-13 页的*文件命名约定*。

1.4 C 和 C++ 库

RVCT 提供以下运行时 C 和 C++ 库：

ARM C 库

ARM C 库提供标准 C 函数以及 C 和 C++ 库使用的辅助函数。C 库还提供依赖于目标的函数，此类函数用于在半主机环境中实现标准 C 库函数，如 `printf`。C 库经过结构化，以便您可以在自己的代码中重新定义依赖于目标的函数来删除半主机相关性。

ARM 库符合：

- 《ARM 体系结构的 C 库 ABI》 (CLIBABI)
- *《ARM 体系结构的 C++ ABI》* (CPPABI)

有关详细信息，请参阅 *《库和浮点支持指南》*中第 1-3 页的*ARM 体系结构 ABI 遵从性*。

Rogue Wave 标准 C++ 库 2.02.03 版

由 Rogue Wave Software, Inc. 提供的 Rogue Wave 标准 C++ 库可提供标准 C++ 函数和对象，如 `cout`。这个库包含称为*标准模板库* (STL) 的数据结构和算法。 C++ 库使用 C 库来提供目标特定的支持。Rogue Wave 标准 C++ 库是在启用 C++ 异常的情况下提供的。

有关 Rogue Wave 库的详细信息，请参阅 Rogue Wave HTML 文档，也可以访问 Rogue Wave 网站：`http://www.roguewave.com`

支持库

ARM C 库提供其他组件以支持 C++，以及为不同体系结构和处理器编译代码。

C 和 C++ 库仅以二进制形式提供。对于主要构建选项的每一种组合，都有一个 1990 ISO 标准 C 库的变体，例如目标系统的字节顺序、是否选中交互操作以及是否选中浮点支持。

有关详细信息，请参阅 *《库和浮点支持指南》*中第 2 章 *C 和 C++ 库*。

第 2 章
编译器命令行选项

本章列出了 ARM 编译器 armcc 所支持的命令行选项。它包括下面一节：

- 第2-2 页的*命令行选项*

2.1 命令行选项

本节按字母顺序列出了编译器所支持的命令行选项。

2.1.1 -A*opt*

此选项指定一些命令行选项，如果编译器在汇编 .s 输入文件或嵌入式汇编语言函数时调用这些命令行选项，它们将传递给汇编器。

语法

-A*opt*

其中：

opt 是要传递给汇编器的命令行选项。

注意

有些编译器命令行选项只要由编译器调用，就会自动传递给汇编器。例如，如果在编译器命令行中指定了 --cpu 选项，则在汇编 .s 文件或嵌入式汇编器时只要调用了此选项，就会将此选项传递给汇编器。

若要查看编译器传递给汇编器的编译器命令行选项，请使用编译器命令行选项 -A--show_cmdline。

示例

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

限制

如果使用 -A 传递了不受支持的选项，则会生成错误。

另请参阅

- 第2-30 页的*--cpu=name*
- 第2-77 页的*-Lopt*
- 第2-112 页的*--show_cmdline*

2.1.2 --allow_null_this, --no_allow_this

此选项允许或禁止在空对象指针上调用成员函数。

缺省设置

缺省选项为 --no_allow_null_this。

2.1.3 --alternative_tokens, --no_alternative_tokens

此选项启用或禁用 C 和 C++ 中备选标记的识别功能。

用法

在 C 和 C++ 中，使用此选项可控制连字的识别。在 C++ 中，使用此选项可控制运算符关键字 （如 **and** 和 **bitand**）的识别。

缺省设置

缺省选项为 --alternative_tokens。

2.1.4 --anachronisms, --no_anachronisms

此选项启用或禁用 C++ 中的过时特性。

模式

仅当源语言为 C++ 时，此选项才有效。

缺省设置

缺省选项为 --no_anachronisms。

示例

```
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
}
```

如果使用 --anachronisms 选项编译此代码，则会生成一条警告消息。

如果在编译此代码时未使用 --anachronisms 选项，则会生成一条错误消息。

另请参阅

- 第2-30 页的*--cpp*
- 第2-115 页的*--strict*, *--no_strict*
- 第2-116 页的*--strict_warnings*
- 第5-14 页的*过时功能*

2.1.5 --apcs=*qualifer...qualifier*

此选项控制生成代码时的交互操作和位置无关性。

通过为 --apcs 命令行选项指定限定符，可以定义编译器所使用的 *《ARM 体系结构的过程调用标准》* (AAPCS) 的变体。

语法

--apcs=*qualifer...qualifier*

其中，*qualifier...qualifier* 表示限定符的列表。该列表必须符合下列条件：

- 至少包含一个限定符
- 其中的各限定符之间没有空格分隔。

qualifier 的每个实例必须为下列值之一：

/interwork、/nointerwork

在支持或不支持 ARM/Thumb™ 交互操作的情况下生成代码。缺省值为 /nointerwork（缺省值为 /interwork 的 ARMv6 及更高版本除外）。

/ropi、/noropi

允许或禁止生成*只读位置无关* (ROPI) 代码。缺省为 /noropi。

/[no]pic 是 /[no]ropi 的别名。

/rwpi、/norwpi

允许或禁止生成*读写且与位置无关* (RWPI) 的代码。缺省为 /norwpi。

/[no]pid 是 /[no]rwpi 的别名。

/fpic、/nofpic　　允许或禁止生成其相对地址引用与程序的加载位置无关的只读且与位置无关的代码。

注意

您也可以指定多个限定符。例如， --apcs=/nointerwork/noropi/norwpi 与 --apcs=/nointerwork --apcs=noropi/norwpi 是等效的。

缺省设置

如果未指定 --apcs 选项，则编译器采用 --apcs=/nointerwork/noropi/norwpi/nofpic。

用法

/interwork, /nointerwork

缺省情况下，按下列方式生成代码：

- 不支持交互操作 （即 /nointerwork），除非指定与体系结构 ARMv5T 或更高版本相对应的 --cpu 选项
- 支持交互操作 （即 /interwork），适用于 ARMv5T 和更高版本。交互操作在 ARMv5T 和更高版本的 ARM 体系结构上自动执行。

/ropi, /noropi　　如果选择使用 /ropi 限定符生成 ROPI 代码，则编译器将执行下列操作：

- 确定只读代码和与 pc 相关的数据的地址
- 在只读输出节上设置 *与位置无关* (PI) 属性。

注意

在编译 C++ 时，不支持 --apcs=/ropi。

/rwpi, /norwpi　　如果选择使用 /rwpi 限定符生成 RWPI 代码，则编译器将执行下列操作：

- 使用相对于静态基址寄存器 sb 的偏移量确定可写数据的地址。这意味着：
  - 可以在运行时固定 RW 数据区域的基址
  - 数据可具有多个实例
  - 数据可以是与位置无关的，但这并不是必需的。
- 在读写输出节上设置 PI 属性。

注意

由于 --lower_rwpi 选项是缺省选项，因此非 RWPI 代码会自动转换为等效的 RWPI 代码。这种静态初始化将在运行时由 C++ 构造函数机制完成，即使在 C 语言中也是如此。

/fpic, /nofpic

如果选择此选项，则编译器将执行下列操作：

- 使用 PC 相对的寻址访问所有静态数据
- 使用链接器所创建的*全局偏移表* (GOT) 条目访问所有导入或导出的读写数据
- 访问与 pc 相关的所有只读数据。

如果代码使用了共享对象，则您必须用 /fpic 编译代码。这是因为仅当代码使用了 System V 共享库时才实现相对寻址。

在构建静态映像或静态库时，无需用 /fpic 进行编译。

在编译 C++ 时支持使用 /fpic。在这种情况下，将把虚拟函数表和 typeinfo 放在读写区域中，以便相对于 PC 的位置对其进行访问。

注意

在生成 System V 或 ARM Linux 共享库时结合使用 --apcs /fpic 和 --no_hide_all。

限制

使用 /ropi、 /rwpi 或 /fpic 编译代码时存在一些限制。

/ropi

用 /ropi 进行编译时，主要存在下列限制：

- 在编译 C++ 时不支持使用 --apcs=/ropi。
- 在用 --apcs=/ropi 进行编译时，一些合法的 C 结构体不起作用。例如：

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

若要使此类静态初始化有效，请使用 --lower_ropi 选项编译代码。例如：

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

/rwpi　　用 /rwpi 进行编译时，主要存在下列限制：

- 在用 --apcs=/rwpi 进行编译时，一些合法的 C 结构体不起作用。例如：

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

若要使此类静态初始化有效，请使用 --lower_rwpi 选项编译代码。例如：

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

注意

因为 --lower_rwpi 是缺省选项，所以不必另行指定。

/fpic　　用 /fpic 进行编译时，主要存在下列限制：

- 如果使用 --apcs=/fpic，则编译器只导出标有 __declspec(dllexport) 的函数和数据。
- 如果在同一命令行中使用 --apcs=/fpic 和 --no_hide_all，则编译器将对所有不使用 __declspec(dll*) 的所有 extern 变量和函数使用缺省 ELF 动态可见性。编译器将对使用缺省 ELF 可见性的函数禁用自动内联功能。
- 如果在 GNU 模式下使用 --apcs=/fpic，还必须使用 --no_hide_all。

另请参阅

- 第2-69 页的*--hide_all, --no_hide_all*
- 第2-85 页的*--lower_ropi, --no_lower_ropi*
- 第2-85 页的*--lower_rwpi, --no_lower_rwpi*
- 第4-24 页的*__declspec(dllexport)*
- *《库和浮点支持指南》*中第2-5 页的*编写可重入且线程安全的代码*
- *《链接器用户指南》*中第3-17 页的*中间代码*
- *《链接器参考指南》*中的第 4 章 *BPABI 和 SysV 共享库和可执行文件*
- *install_directory*\Documentation\Specifications\... 中的 《ARM 体系结构的过程调用标准》

2.1.6 --arm

此选项表示请求编译器以 ARM 指令集为目标。编译器可以生成 ARM 和 Thumb 代码，但首选识别 ARM 代码。

注意

此选项不适用于只支持 Thumb 的处理器 （如 Cortex-M3）。

缺省设置

这是支持 ARM 指令集的目标的缺省选项。

另请参阅

- 第2-14 页的*--arm_only*
- 第2-30 页的*--cpu=list*
- 第2-30 页的*--cpu=name*
- 第2-118 页的*--thumb*
- 第4-56 页的*#pragma arm*
- *《编译器用户指南》*中第2-23 页的*指定目标处理器或体系结构*。

2.1.7 --arm_linux

此选项以适用于 ARM Linux 编译的缺省值配置一组其他选项。

用法

当使用以下某个 ARM Linux 选项时，这些缺省值会自动启用：

- --arm_linux_paths
- 完整 GCC 仿真模式中的 --translate_gcc
- 完整 GCC 仿真模式中的 --translate_g++
- 完整 GCC 仿真模式中的 --translate_gld

此选项通常用于帮助移植旧代码。它用于简化现有 makefile 中使用的编译器选项，同时保留对所用头文件和库搜索路径的全面显式控制。

从 RVCT 4.0 版之前的版本移植时，您可以将所有这些提供给编译器的选项替换为一个 --arm_linux 选项。

缺省设置

缺省情况下，配置的这组选项包括：

- --apcs=/interwork
- --enum_is_int
- --gnu
- --library_interface=aeabi_glibc
- --no_hide_all
- --preinclude=linux_rvct.h
- --wchar32

示例

若要应用缺省选项集，请使用 --arm_linux。

若要覆盖任何缺省选项，请分别进行指定。例如，--arm_linux --hide_all。

在后面的示例中，前面带有 --arm_linux 的 --hide_all 覆盖了 --no_hide_all。

另请参阅

- 第2-10 页的*--arm_linux_config_file=path*
- 第2-11 页的*--arm_linux_configure*
- 第2-13 页的*--arm_linux_paths*
- 第2-24 页的*--configure_cpp_headers=path*
- 第2-25 页的*--configure_extra_includes=paths*
- 第2-26 页的*--configure_extra_libraries=paths*
- 第2-27 页的*--configure_gcc=path*
- 第2-28 页的*--configure_gld=path*
- 第2-29 页的*--configure_sysroot=path*
- 第2-111 页的*--shared*
- 第2-118 页的*--translate_g++*
- 第2-120 页的*--translate_gcc*
- 第2-121 页的*--translate_gld*
- *《链接器参考指南》*中第2-3 页的*--arm_linux*

- 《链接器参考指南》中第2-33 页的*--library=name*
- 《链接器参考指南》中第2-51 页的*--[no_]search_dynamic_libraries*
- 《应用程序注释 212 — 使用 RVCT 4.0 版以及 GNU 工具和库生成 Linux 应用程序》(*Application Note 212 - Building Linux applications using RVCT v4.0 and the GNU Tools and Libraries*)。

2.1.8 --arm_linux_config_file=*path*

此选项指定为 ARM Linux 生成而创建的配置文件的位置。它允许在编译代码时使用标准 Linux 配置设置。

语法

--arm_linux_config_file=*path*

其中 *path* 是配置文件的路径和文件名。

限制

必须在生成配置文件时以及在编译和链接期间使用配置时都使用此选项。

如果在命令行中指定 ARM Linux 配置文件并使用 --translate_gcc、--translate_g++ 或 --translate_gld，则会影响某些其他选项的缺省设置。--bss_threshold 的缺省值变为零，--signed_bitfields 和 --unsigned_bitfields 的缺省值变为 --signed_bitfields，并启用 --enum_is_int 和 --wchar32。

另请参阅

- 第2-8 页的*--arm_linux*
- 第2-11 页的*--arm_linux_configure*
- 第2-13 页的*--arm_linux_paths*
- 第2-20 页的*--bss_threshold=num*
- 第2-24 页的*--configure_cpp_headers=path*
- 第2-25 页的*--configure_extra_includes=paths*
- 第2-26 页的*--configure_extra_libraries=paths*
- 第2-27 页的*--configure_gcc=path*
- 第2-28 页的*--configure_gld=path*
- 第2-29 页的*--configure_sysroot=path*

- 第2-52 页的*--enum_is_int*
- 第2-111 页的*--shared*
- 第2-112 页的*--signed_bitfields*, *--unsigned_bitfields*
- 第2-118 页的*--translate_g++*
- 第2-120 页的*--translate_gcc*
- 第2-121 页的*--translate_gld*
- 第2-131 页的*--wchar32*
- *《链接器参考指南》*中第2-3 页的*--arm_linux*
- *《链接器参考指南》*中第2-33 页的*--library=name*
- *《链接器参考指南》*中第2-51 页的*--[no_]search_dynamic_libraries*
- 《应用程序注释 212 — 使用 RVCT 4.0 版以及 GNU 工具和库生成 Linux 应用程序》(*Application Note 212 - Building Linux applications using RVCT v4.0 and the GNU Tools and Libraries*)。

2.1.9 --arm_linux_configure

此选项通过创建描述包含路径、库路径和 GNU C 库 glibc 的标准库的配置文件，来配置用于 ARM Linux 的 RVCT。生成代码时将使用创建的配置文件。

用法

应用自动和手动配置方法。自动配置尝试自动在 PATH 环境变量中查找 GNU 工具链的安装，并对其进行查询以确定要使用的配置设置。手动配置可用于自行指定头文件和库的位置。如果未安装完整的 GNU 工具链，可以使用手动配置。

执行自动配置：

- armcc --arm_linux_configure --arm_linux_config_file=*config_file_path* --configure_gcc=*path* --configure_gld=*path*

  其中 *config_file_path* 是所创建的配置文件的路径和文件名。您可以选择指定 *GNU 编译器集合* (GCC) 驱动程序的位置，也可以选择指定 GNU 链接器的位置，来覆盖系统 PATH 环境变量确定的位置。

执行手动配置：

- armcc --arm_linux_configure --arm_linux_config_file=*path* --configure_cpp_headers=*path* --configure_sysroot=*path*

  其中指定了 GNU libstdc++ *标准模板库* (STL) 头文件的路径，以及库和头文件所在的系统根路径。

限制

系统中必须存在 GNU 工具链才能使用自动配置。

如果使用自动配置方法，则必须使用系统 PATH 环境变量来查找 ARM Linux GCC。如果系统路径中没有合适的 GCC，可以在路径中添加一个，也可以使用 --configure_gcc （并选择使用 --configure_gld）手动指定合适的 GCC 的位置。

缺省设置

除非使用其他选项指定 GCC 或 GNU 链接器的位置，否则会应用自动配置。也就是说，除非使用其他选项指定 ARM Linux GCC 的位置，否则编译器会尝试使用系统路径环境变量查找 ARM Linux GCC。

另请参阅

- 第2-8 页的*--arm_linux*
- 第2-10 页的*--arm_linux_config_file=path*
- 第2-13 页的*--arm_linux_paths*
- 第2-27 页的*--configure_gcc=path*
- 第2-28 页的*--configure_gld=path*
- 第2-29 页的*--configure_sysroot=path*
- 第2-24 页的*--configure_cpp_headers=path*
- 第2-25 页的*--configure_extra_includes=paths*
- 第2-26 页的*--configure_extra_libraries=paths*
- 第2-111 页的*--shared*
- 第2-118 页的*--translate_g++*
- 第2-120 页的*--translate_gcc*
- 第2-121 页的*--translate_gld*
- *《链接器参考指南》*中第2-3 页的*--arm_linux*
- *《链接器参考指南》*中第2-33 页的*--library=name*
- *《链接器参考指南》*中第2-51 页的*--[no_]search_dynamic_libraries*
- 《应用程序注释 212 — 使用 RVCT 4.0 版以及 GNU 工具和库生成 Linux 应用程序》(*Application Note 212 - Building Linux applications using RVCT v4.0 and the GNU Tools and Libraries*)。

2.1.10 --arm_linux_paths

使用此选项可以为 ARM Linux 生成代码。

用法

配置了用于 ARM Linux 的 RVCT 后，可以使用此选项。

这只是编译器选项。它遵循典型的 GCC 使用模型，即编译器驱动程序用于标准系统对象文件和库的直接链接和选择。

此选项还可用于帮助从 RVCT 4.0 版之前的 RVCT 版本进行迁移。使用 --arm_linux_configure 创建件配置文件后，可以通过替换标准选项列表来修改现有生成，并用 --arm_linux_paths 选项搜索路径。也就是说，--arm_linux_paths 可用于替换：

- 为 --arm_linux 列出的所有缺省选项
- 头文件路径
- 库路径
- 标准库。

限制

必须使用 --arm_linux_config_file=*filename* 指定配置文件的位置。

示例

编译并链接应用程序代码：

```
armcc --arm_linux_paths --arm_linux_config_file=my_config_file -o hello -O2
-Otime -g hello.c
```

编译源文件 source.c 以便在共享库中使用：

```
armcc --arm_linux_paths --arm_linux_config_file=my_config_file --apcs=/fpic -c
source.c
```

使用编译器将两个对象文件 obj1 和 obj2 链接到共享库 my_shared_lib.so 中：

```
armcc --arm_linux_paths --arm_linux_config_file=my_config_file --shared -o
my_shared_lib.so obj1.o obj2.o
```

另请参阅

- 第2-8 页的*--arm_linux*
- 第2-10 页的*--arm_linux_config_file=path*
- 第2-11 页的*--arm_linux_configure*
- 第2-27 页的*--configure_gcc=path*
- 第2-28 页的*--configure_gld=path*
- 第2-29 页的*--configure_sysroot=path*
- 第2-24 页的*--configure_cpp_headers=path*
- 第2-25 页的*--configure_extra_includes=paths*
- 第2-26 页的*--configure_extra_libraries=paths*
- 第2-111 页的*--shared*
- 第2-118 页的*--translate_g++*
- 第2-120 页的*--translate_gcc*
- 第2-121 页的*--translate_gld*
- *《链接器参考指南》*中第2-3 页的*--arm_linux*
- *《链接器参考指南》*中第2-33 页的*--library=name*
- *《链接器参考指南》*中第2-51 页的*--[no_]search_dynamic_libraries*
- 《应用程序注释 212 — 使用 RVCT 4.0 版以及 GNU 工具和库生成 Linux 应用程序》(*Application Note 212 - Building Linux applications using RVCT v4.0 and the GNU Tools and Libraries*)。

2.1.11 --arm_only

此选项强制只编译 ARM 代码。编译器就当目标体系结构中没有 Thumb 一样进行工作。

编译器将 --arm_only 选项传播到汇编器和链接器。

示例

```
armcc --arm_only myprog.c
```

注意

如果指定 armcc --arm_only --thumb myprog.c，并不意味着编译器会检查代码以确保不存在 Thumb 代码。这意味着 --thumb 将覆盖 --arm_only。

另请参阅

- 第2-8 页的*--arm*
- 第2-118 页的*--thumb*
- 有关 --16 和 --32 的信息，请参阅 *《汇编器指南》*中第3-2 页的*命令语法*。

2.1.12 --asm

此选项指示编译器将列表写入编译器所生成的机器代码的反汇编文件中。

如果选择此选项，则会生成对象代码。此外，除非选择了 -c 选项，否则还会执行链接步骤。

注意

若要在不生成对象代码的情况下对编译器生成的机器代码执行反汇编，请选择 -S 而不是 --asm。

用法

--asm 的操作以及所生成的反汇编文件的完整名称因所使用的选项组合而异：

表 2-1 用 --asm 选项进行编译

编译器选项	操作
--asm	将一个列表写入编译源的反汇编文件。 此外，除非使用了 c 选项，否则还会执行链接步骤。 反汇编代码将写入一个文本文件中，其缺省名称为具有文件扩展名 .s 的输入文件的名称。
--asm -c	与 --asm 类似，但不执行链接步骤。

表 2-1 用 --asm 选项进行编译 （续）

编译器选项	操作
--asm --interleave	与 --asm 类似，但源代码将与反汇编代码进行交叉存取。 反汇编代码将写入一个文本文件中，其缺省名称为具有文件扩展名 .txt 的输入文件的名称。
--asm --multifile	与 --asm 类似，但编译器将为合并到主文件中的文件生成空对象文件。
--asm -o *filename*	与 --asm 类似，但对象文件将命名为 *filename*。 反汇编代码将写入文件 *filename.s* 中。 对象文件的名称中不能包含文件扩展名 .s。如果对象文件的文件扩展名为 .s，则会将反汇编代码写入对象文件的最上方。这可能会导致不可预知的结果。

另请参阅

- 第2-21 页的*-c*
- 第2-75 页的*--interleave*
- 第2-90 页的*--multifile*, *--no_multifile*
- 第2-92 页的*-o filename*
- 第2-110 页的*-S*
- *《编译器用户指南》*中第2-13 页的*文件命名约定*。

2.1.13 --autoinline, --no_autoinline

此选项启用或禁用函数的自动内联。

编译器会在合理的情况下针对较高优化级别自动内联函数。-Ospace 和 -Otime 选项以及其他某些因素 （如函数大小）会对编译器自动内联函数的方式产生影响。

通过选择 -Otime 并结合各种其他因素，会增加内联函数的可能性。

缺省设置

对于优化级别 -O0 和 -O1，缺省为 --no_autoinline。

对于优化级别 -O2 和 -O3，缺省为 --autoinline。

另请参阅

- 第2-58 页的*--forceinline*
- 第2-73 页的*--inline*, *--no_inline*
- 第2-94 页的*-Onum*
- 第2-96 页的*-Ospace*
- 第2-96 页的*-Otime*

2.1.14 --bigend

此选项指示编译器使用大端内存为 ARM 处理器生成代码。

ARM 体系结构定义以下两种不同的大端模式：

BE8 字节固定寻址模式 （ARMv6 及更高版本）。

BE32 旧大端模式。

在链接时指定是选择 BE8 还是 BE32。

缺省设置

除非显式指定 --bigend，否则编译器将采用 --littleend。

另请参阅

- 第2-83 页的*--littleend*
- *《开发指南》*中第2-13 页的*端支持*
- *《链接器参考指南》*中第2-4 页的*--be8*
- *《链接器参考指南》*中第2-5 页的*--be32*

2.1.15 --bitband

此选项将对所有非 **const** 全局结构对象执行位处理操作。使用此选项，可以将一个内存字映射到位处理操作区中的单个位。这将实现对内存体系结构的 SRAM 和外设区中的单个位值进行有效的原子访问。

对于区分内存访问宽度的外设，分别为位处理操作结构的位域的 **char**、**short** 和 **int** 类型生成存储或加载到别名空间的字节、半字和字。

限制

应用的限制如下：

- 此选项只影响 **struct** 类型。任何联合类型或其他带有联合成员的聚合类型都不能进行位处理操作。
- 结构中的成员无法单独进行位处理操作。
- 仅为包含单个位的位域生成位处理操作访问。
- 不对 **const** 对象、指针和局部对象生成位处理操作访问。

示例

在示例 2-1 中，当使用 --bitband 命令行选项进行编译时，将在写入位域 i 和 k 时执行位处理操作。

示例 2-1 位处理操作示例

```
typedef struct {
  int i : 1;
  int j : 2;
  int k : 1;
} BB;

BB value;

void update_value(void)
{
  value.i = 1;
  value.k = 1;
}
```

另请参阅

- 第4-42 页的*__attribute__((bitband))*
- 《编译器用户指南》中第4-15 页的*位处理操作*
- 处理器的 《技术参考手册》 (*Technical Reference Manual*)。

2.1.16 --brief_diagnostics, --no_brief_diagnostics

此选项允许或禁止输出编译器所生成的简短诊断消息。

如果启用此选项，则不显示原始源代码行，并且当错误消息文本太长而在一行放不下时也不换行。

缺省设置

缺省选项为 --no_brief_diagnostics。

示例

```
/* main.c */
#include <stdio.h>
int main(void)
{
   printf(" "Hello, world\n");
   return 0;
}
```

用 --brief_diagnostics 选项编译此代码时会生成一条警告消息。

另请参阅

- 第2-44 页的*--diag_error=tag[,tag,...]*
- 第2-45 页的*--diag_remark=tag[,tag,...]*
- 第2-45 页的*--diag_style={arm|ide|gnu}*
- 第2-46 页的*--diag_suppress=tag[,tag,...]*
- 第2-48 页的*--diag_warning=tag[,tag,...]*
- 第2-52 页的*--errors=filename*
- 第2-108 页的*--remarks*
- 第2-129 页的*-W*
- 第2-133 页的*--wrap_diagnostics*, *--no_wrap_diagnostics*
- *《编译器用户指南》*中的第 6 章 *诊断消息*。

2.1.17 --bss_threshold=*num*

此选项控制小型全局 ZI 数据项在节中的放置。*小型全局 ZI 数据项*是小于或等于八字节的未初始化数据项。

语法

--bss_threshold=*num*

其中：

num 是下列值之一：

- 0 将小型全局 ZI 数据项放在 ZI 数据节中
- 8 将小型全局 ZI 数据项放在 RW 数据节中。

用法

在 RVCT 的当前版本中，编译器可在优化时将小型全局 ZI 数据项放置在 RW 数据节中。在 RVCT 2.0.1 及更低版本中，缺省情况下会将小型全局 ZI 数据项放置在 ZI 数据节中。

使用此选项可模拟 RVCT 2.0.1 及更低版本中与小型全局 ZI 数据项在 ZI 数据节中的放置有关的行为。

注意

选择 --bss_threshold=0 选项时，可指示编译器将*所有*小型全局 ZI 数据项放置在 ZI 数据节的当前编译模块中。若要将特定变量放置在：

- 某一 ZI 数据节中，请使用 __attribute__((zero_init))
- 特定的 ZI 数据节中，请组合使用 __attribute__((section)) 和 __attribute__((zero_init))。

缺省设置

如果未指定 --bss_threshold 选项，编译器将采用 --bss_threshold=8。

如果在命令行中指定 ARM Linux 配置文件并使用 --translate_gcc 或 --translate_g++，编译器将采用 --bss_threshold=0。

示例

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

使用 --bss_threshold=0 编译此代码时，会将 glob1 放置在 ZI 数据节中。

另请参阅

- 第4-56 页的*#pragma arm section [section_sort_list]*
- 第2-10 页的*--arm_linux_config_file=path*
- 第2-11 页的*--arm_linux_configure*
- 第4-50 页的*__attribute__((section("name")))*
- 第4-54 页的*__attribute__((zero_init))*

2.1.18 -c

此选项指示编译器执行编译步骤，但不执行链接步骤。

注意

此选项不同于大写的 -C 选项。

用法

建议在包含多个源文件的项目中使用 -c 选项。

另请参阅

- 第2-15 页的*--asm*
- 第2-80 页的*--list*
- 第2-92 页的*-o filename*
- 第2-110 页的*-S*

2.1.19 -C

此选项指示编译器在预处理程序输出中保留注释。

选择此选项会隐式选择 -E 选项。

注意

此选项不同于小写的 -c 选项。

另请参阅

- 第2-51 页的*-E*

2.1.20 --c90

此选项启用 C90 源代码的编译。

缺省设置

系统会为具有 .c、.ac 或 .tc 后缀的文件隐式选择此选项。

注意

不再使用文件扩展名 .ac 和 .tc。

另请参阅

- *--c99*
- 第2-66 页的*--gnu*
- 第2-115 页的*--strict*, *--no_strict*
- 第1-3 页的*源语言模式*
- *《编译器用户指南》*中第2-13 页的*文件命名约定*。

2.1.21 --c99

此选项启用 C99 源代码的编译。

另请参阅

- *--c90*
- 第2-66 页的*--gnu*
- 第2-115 页的*--strict*, *--no_strict*
- 第1-3 页的*源语言模式*

2.1.22 --code_gen, --no_code_gen

此选项允许或禁止生成对象代码。

如果禁止生成对象代码，则编译器只执行语法检查，而不创建对象文件。

缺省设置

缺省选项为 --code_gen。

2.1.23 --compatible=*name*

此选项可让编译器生成与多种处理器或体系结构兼容的代码。

表 2-2 中显示了这些有效组合。您可以将组 1 中的任意处理器或体系结构与组 2 中的任意处理器或体系结构匹配。

表 2-2 兼容的处理器或体系结构组合

组 1	ARM7TDMI, 4T
组 2	Cortex-M1, Cortex-M3, 7-M, 6-M, 6S-M

语法

--compatible=*name*

其中：

name 是处理器或体系结构的名称，或 NONE。处理器和体系结构名称都不区分大小写。

如果命令行中存在此选项的多个实例，则最后指定的实例会覆盖前面的实例。

在命令行末尾指定 --compatible=NONE 可关闭此选项的所有其他实例。

示例

```
armcc --cpu=arm7tdmi --compatible=cortex-m3 myprog.c
```

另请参阅

- 第2-30 页的*--cpu=name*

2.1.24 --compile_all_input, --no_compile_all_input

此选项启用或禁用文件扩展名处理的禁止行为。

如果启用该禁止行为，则编译器将完全禁止文件扩展名的处理，同时将所有输入文件视为具有后缀 .c。

缺省设置

缺省选项为 --no_compile_all_input。

另请参阅

- *《编译器用户指南》*中第2-13 页的*文件命名约定*。

2.1.25 --configure_cpp_headers=*path*

当配置要用于 ARM Linux 的 RVCT 时，此选项指定 GNU libstdc++ STL 头文件的路径。

语法

--configure_cpp_headers=*path*

其中：

path 是 GNU C++ STL 头文件的路径。

用法

此选项会覆盖自动检测到的任何路径。它可作为手动方法的一部分来配置用于 ARM Linux 的 RVCT。

另请参阅

- 第2-8 页的*--arm_linux*
- 第2-10 页的*--arm_linux_config_file=path*
- 第2-11 页的*--arm_linux_configure*
- 第2-13 页的*--arm_linux_paths*
- 第2-27 页的*--configure_gcc=path*
- 第2-28 页的*--configure_gld=path*
- 第2-29 页的*--configure_sysroot=path*
- 第2-25 页的*--configure_extra_includes=paths*
- 第2-26 页的*--configure_extra_libraries=paths*
- 第2-111 页的*--shared*
- 第2-118 页的*--translate_g++*
- 第2-120 页的*--translate_gcc*
- 第2-121 页的*--translate_gld*

- 《链接器参考指南》中第2-3 页的*--arm_linux*
- 《链接器参考指南》中第2-33 页的*--library=name*
- 《链接器参考指南》中第2-51 页的*--[no_]search_dynamic_libraries*
- 《应用程序注释 212 — 使用 RVCT 4.0 版以及 GNU 工具和库生成 Linux 应用程序》(*Application Note 212 - Building Linux applications using RVCT v4.0 and the GNU Tools and Libraries*)。

2.1.26 --configure_extra_includes=*paths*

当配置要用于 ARM Linux 的 RVCT 时，此选项指定所有附加系统包含路径。

语法

--configure_extra_includes=*paths*

其中：

paths 是以逗号分隔的路径名列表，表示附加系统包含路径的位置。

另请参阅

- 第2-8 页的*--arm_linux*
- 第2-10 页的*--arm_linux_config_file=path*
- 第2-11 页的*--arm_linux_configure*
- 第2-13 页的*--arm_linux_paths*
- 第2-24 页的*--configure_cpp_headers=path*
- 第2-26 页的*--configure_extra_libraries=paths*
- 第2-27 页的*--configure_gcc=path*
- 第2-28 页的*--configure_gld=path*
- 第2-29 页的*--configure_sysroot=path*
- 第2-111 页的*--shared*
- 第2-118 页的*--translate_g++*
- 第2-120 页的*--translate_gcc*
- 第2-121 页的*--translate_gld*
- 《链接器参考指南》中第2-3 页的*--arm_linux*
- 《链接器参考指南》中第2-33 页的*--library=name*
- 《链接器参考指南》中第2-51 页的*--[no_]search_dynamic_libraries*

- 《应用程序注释 212 — 使用 RVCT 4.0 版以及 GNU 工具和库生成 Linux 应用程序》(*Application Note 212 - Building Linux applications using RVCT v4.0 and the GNU Tools and Libraries*)。

2.1.27 --configure_extra_libraries=*paths*

当配置要用于 ARM Linux 的 RVCT 时，此选项指定所有附加系统库路径。

语法

`--configure_extra_libraries=`*paths*

其中：

paths 是以逗号分隔的路径名列表，表示附加系统库路径的位置。

另请参阅

- 第2-8 页的*--arm_linux*
- 第2-10 页的*--arm_linux_config_file=path*
- 第2-11 页的*--arm_linux_configure*
- 第2-13 页的*--arm_linux_paths*
- 第2-24 页的*--configure_cpp_headers=path*
- 第2-25 页的*--configure_extra_includes=paths*
- 第2-27 页的*--configure_gcc=path*
- 第2-28 页的*--configure_gld=path*
- 第2-29 页的*--configure_sysroot=path*
- 第2-111 页的*--shared*
- 第2-118 页的*--translate_g++*
- 第2-120 页的*--translate_gcc*
- 第2-121 页的*--translate_gld*
- *《链接器参考指南》*中第2-3 页的*--arm_linux*
- *《链接器参考指南》*中第2-33 页的*--library=name*
- *《链接器参考指南》*中第2-51 页的*--[no_]search_dynamic_libraries*
- 《应用程序注释 212 — 使用 RVCT 4.0 版以及 GNU 工具和库生成 Linux 应用程序》(*Application Note 212 - Building Linux applications using RVCT v4.0 and the GNU Tools and Libraries*)。

2.1.28 --configure_gcc=*path*

当配置要用于 ARM Linux 的 RVCT 时，此选项指定 GCC 驱动程序的位置。

语法

`--configure_gcc=`*path*

其中：

path 是 GCC 驱动程序的路径和文件名。

用法

如果要覆盖配置时指定的 GCC 驱动程序缺省位置，或者 `--arm_linux_configure` 的自动配置方法无法找到驱动程序，可以使用此选项。

另请参阅

- 第2-8 页的*--arm_linux*
- 第2-10 页的*--arm_linux_config_file=path*
- 第2-11 页的*--arm_linux_configure*
- 第2-13 页的*--arm_linux_paths*
- 第2-24 页的*--configure_cpp_headers=path*
- 第2-25 页的*--configure_extra_includes=paths*
- 第2-26 页的*--configure_extra_libraries=paths*
- 第2-28 页的*--configure_gld=path*
- 第2-29 页的*--configure_sysroot=path*
- 第2-111 页的*--shared*
- 第2-118 页的*--translate_g++*
- 第2-120 页的*--translate_gcc*
- 第2-121 页的*--translate_gld*
- *《链接器参考指南》*中第2-3 页的*--arm_linux*
- *《链接器参考指南》*中第2-33 页的*--library=name*
- *《链接器参考指南》*中第2-51 页的*--[no_]search_dynamic_libraries*
- 《应用程序注释 212 — 使用 RVCT 4.0 版以及 GNU 工具和库生成 Linux 应用程序》(*Application Note 212 - Building Linux applications using RVCT v4.0 and the GNU Tools and Libraries*)。

2.1.29 --configure_gld=*path*

此选项指定 GNU 链接器 ld 的位置。

语法

--configure_gld=*path*

其中：

path 是 GNU 链接器的路径和文件名。

用法

配置时，编译器会尝试确定 GCC 使用的 GNU 链接器的位置。如果编译器无法确定位置，或者要覆盖 GNU 链接器的常规路径，可以使用 --configure_gld=*path* 选项指定其位置。该路径是 GNU ld 二进制文件的完整路径和文件名。

另请参阅

- 第2-8 页的*--arm_linux*
- 第2-10 页的*--arm_linux_config_file=path*
- 第2-11 页的*--arm_linux_configure*
- 第2-13 页的*--arm_linux_paths*
- 第2-24 页的*--configure_cpp_headers=path*
- 第2-25 页的*--configure_extra_includes=paths*
- 第2-26 页的*--configure_extra_libraries=paths*
- 第2-27 页的*--configure_gcc=path*
- 第2-29 页的*--configure_sysroot=path*
- 第2-111 页的*--shared*
- 第2-118 页的*--translate_g++*
- 第2-120 页的*--translate_gcc*
- 第2-121 页的*--translate_gld*
- *《链接器参考指南》*中第2-3 页的*--arm_linux*
- *《链接器参考指南》*中第2-33 页的*--library=name*
- *《链接器参考指南》*中第2-51 页的*--[no_]search_dynamic_libraries*

- 《应用程序注释 212 — 使用 RVCT 4.0 版以及 GNU 工具和库生成 Linux 应用程序》(*Application Note 212 - Building Linux applications using RVCT v4.0 and the GNU Tools and Libraries*)。

2.1.30 --configure_sysroot=*path*

当配置要用于 ARM Linux 的 RVCT 时，此选项指定要使用的系统根路径。

语法

`--configure_sysroot=`*path*

其中 *path* 是要使用的系统根路径。

用法

此选项会覆盖自动检测到的任何系统根路径。如果要使用其他路径作为常规系统根路径，它可作为手动方法的一部分来配置用于 ARM Linux 的 RVCT。

系统根路径是通常存放库和头文件的基准路径。在标准 Linux 系统上，这通常是文件系统的根。在交叉编译 GNU 工具链中，它通常是 GNU C 库安装的父目录。此目录包含存放 C 库和头文件的 `lib`、`usr/lib` 和 `usr/include` 子目录。

另请参阅

- 第2-8 页的*--arm_linux*
- 第2-10 页的*--arm_linux_config_file=path*
- 第2-11 页的*--arm_linux_configure*
- 第2-13 页的*--arm_linux_paths*
- 第2-24 页的*--configure_cpp_headers=path*
- 第2-25 页的*--configure_extra_includes=paths*
- 第2-26 页的*--configure_extra_libraries=paths*
- 第2-27 页的*--configure_gcc=path*
- 第2-28 页的*--configure_gld=path*
- 第2-111 页的*--shared*
- 第2-118 页的*--translate_g++*
- 第2-120 页的*--translate_gcc*
- 第2-121 页的*--translate_gld*
- *《链接器参考指南》*中第2-3 页的*--arm_linux*
- *《链接器参考指南》*中第2-33 页的*--library=name*

- 《链接器参考指南》中第2-51 页的*--[no_]search_dynamic_libraries*
- 《应用程序注释 212 — 使用 RVCT 4.0 版以及 GNU 工具和库生成 Linux 应用程序》(*Application Note 212 - Building Linux applications using RVCT v4.0 and the GNU Tools and Libraries*)。

2.1.31 --cpp

此选项启用 C++ 源代码的编译。

缺省设置

系统会为具有 .cpp、.cxx、.c++、.cc 或 .CC 后缀的文件隐式选择此选项。

另请参阅

- 第2-3 页的*--anachronisms*, *--no_anachronisms*
- 第2-22 页的*--c90*
- 第2-22 页的*--c99*
- 第2-66 页的*--gnu*
- 第2-115 页的*--strict*, *--no_strict*
- 第1-3 页的*源语言模式*

2.1.32 --cpu=list

此选项列出可用于 --cpu=*name* 选项的受支持体系结构和处理器的名称。

另请参阅

- *--cpu=name*

2.1.33 --cpu=*name*

此选项为所选的 ARM 处理器或体系结构启用代码生成。

语法

--cpu=*name*

其中：

name 是处理器或体系结构的名称。

如果 *name* 是处理器的名称，则按 ARM 数据手册中所示的形式输入该名称；例如， ARM7TDMI、 ARM1176JZ-S、 MPCore。

如果 *name* 是体系结构的名称，则该名称必须包含在表 2-3 中所示的体系结构列表内。

处理器和体系结构名称都不区分大小写。

不接受通配符。

表 2-3 支持的 ARM 体系结构和示例处理器

体系结构	说明	示例处理器
4	没有 Thumb 的 ARMv4	SA-1100
4T	具有 Thumb 的 ARMv4	ARM7TDMI、 ARM9TDMI、 ARM720T、 ARM740T、 ARM920T、 ARM922T、 ARM940T、 SC100
5T	具有 Thumb 和交互操作的 ARMv5	
5TE	具有 Thumb、交互操作、 DSP 乘法以及双字指令的 ARMv5	ARM9E、 ARM946E-S、 ARM966E-S
5TEJ	具有 Thumb、交互操作、 DSP 乘法、双字指令以及 Jazelle® 扩展[a] 的 ARMv5	ARM926EJ-S、 ARM1026EJ-S、 SC200
6	具有 Thumb、交互操作、 DSP 乘法、双字指令、未对齐和混合端支持、 Jazelle 扩展以及多媒体扩展的 ARMv6	ARM1136J-S、 ARM1136JF-S
6-M	ARMv6 微控制器规格，仅具有 Thumb 并增加了处理器状态指令	无操作系统扩展的 Cortex-M1
6S-M	ARMv6 微控制器规格，仅具有 Thumb 并增加了处理器状态指令和操作系统扩展	具有操作系统扩展的 Cortex-M1
6K	具有 SMP 扩展的 ARMv6	MPCore
6T2	具有 Thumb-2 的 ARMv6	ARM1156T2-S、 ARM1156T2F-S

表 2-3 支持的 ARM 体系结构和示例处理器 （续）

体系结构	说明	示例处理器
6Z	具有安全扩展的 ARMv6	ARM1176JZF-S、ARM1176JZ-S
7	仅具有 Thumb-2 且没有硬件除法器的 ARMv7	
7-A	支持基于虚拟 MMU 的内存系统的 ARMv7 应用程序规格，具有 ARM、Thumb-2 和 Thumb-2EE 指令集、DSP 支持以及 32 位 SIMD 支持	Cortex-A8、 Cortex-A9
7-R	ARMv7 实时规格，具有 ARM、 Thumb-2、 DSP 支持以及 32 位 SIMD 支持	Cortex-R4、Cortex-R4F
7-M	ARMv7 微控制器规格，仅具有 Thumb-2 且具有硬件除法器	Cortex-M3、 SC300

a. ARM 编译器不能生成 Java 字节代码。

注意

ARMv7 不是实际的 ARM 体系结构。`--cpu=7` 指的是所有 ARMv7-A、ARMv7-R 和 ARMv7-M 体系结构都具有的功能。根据定义， `--cpu=7` 选项所具有的任何给定功能在所有 ARMv7-A、 ARMv7-R 和 ARMv7-M 体系结构中都存在。

缺省设置

如果未指定 `--cpu` 选项，编译器将采用 `--cpu=ARM7TDMI`。

若要获取 CPU 体系结构和处理器的完整列表，请使用 `--cpu=list` 选项。

用法

以下主要的几点适用于处理器和体系结构选项：

处理器

- 选择处理器时需要选择适当的体系结构、*浮点单元* (FPU) 以及内存组织。
- 受支持的 `--cpu` 值包括所有当前的 ARM 产品名称或体系结构版本。

  此外，还支持其他基于 ARM 体系结构的处理器，如 Marvell Feroceon 和 Intel XScale。

- 如果为 --cpu 选项指定了处理器，则将针对该处理器优化已编译的代码。这使得编译器可使用特定的协处理器或指令调度来优化性能。

体系结构

- 如果为 --cpu 选项指定了体系结构名称，则会编译代码，使其可在支持该体系结构的任何处理器上运行。例如，--cpu=5TE 所生成的代码可由 ARM926EJ-S® 使用。

FPU

- 有些 --cpu 规范中隐含 --fpu 的选择。例如，在用 --arm 选项进行编译时，--cpu=ARM1136JF-S 隐含了 --fpu=vfpv2。类似地，--cpu=Cortex-R4F 隐含了 --fpu=vfpv3_d16。

  注意

  在命令行中用 --fpu 设置的任意*显式*FPU 都将覆盖*隐式*FPU。

- 如果既未指定 --fpu 选项也未指定 --cpu 选项，则会使用 --fpu=softvfp。

ARM/Thumb

- 指定支持 Thumb 指令的处理器或体系结构 （如 --cpu=ARM7TDMI）时，并不会使编译器生成 Thumb 代码。该操作只启用要使用的处理器的功能，如长乘法。使用 --thumb 选项可生成 Thumb 代码，除非处理器是仅支持 Thumb 的处理器，如 Cortex-M3。在此情况下， --thumb 不是必需的。

  注意

  指定目标处理器或体系结构时，可能会导致编译器所生成的对象代码与其他 ARM 处理器不兼容。例如，如果为体系结构 ARMv6 编译的代码包含 ARMv6 特有的指令，则这些代码可能无法在 ARM920T 处理器上运行。因此，必须选择同时符合您各种需要的处理器。

- 如果要为支持 ARMv4T 或 ARMv5T 的处理器编译专用于 ARM/Thumb 混合系统的代码，则必须指定交互操作选项 --apcs=/interwork。缺省情况下，将为支持 ARMv5T 或更高版本的处理器启用此选项。
- 如果要为 Thumb 编译代码 （即在命令行中使用 --thumb 选项进行编译），则编译器会尽可能多地使用 Thumb 指令集编译代码。但编译器可能会在编译过程中生成 ARM 代码。例如，如果要为 Thumb-1 处理器编译代码并使用了 VFP，则将为 ARM 编译包含浮点运算的任何函数。

- 如果要为 ARMv7-M （如 --cpu=Cortex-M3）编译代码，则无需在命令行中指定 --thumb，因为 ARMv7-M 仅支持 Thumb-2。

限制

不能在同一命令行中同时指定处理器和体系结构。

另请参阅

- 第2-4 页的*--apcs=qualifer...qualifier*
- 第2-30 页的*--cpu=list*
- 第2-61 页的*--fpu=name*
- 第2-118 页的*--thumb*

2.1.34 --create_pch=*filename*

此选项指示编译器创建一个具有指定文件名的*预编译头文件* (PCH) 文件。

此选项优先于所有其他的 PCH 选项。

语法

--create_pch=*filename*

其中：

filename 是要创建的 PCH 文件的名称。

另请参阅

- 第2-98 页的*--pch*
- 第2-98 页的*--pch_dir=dir*
- 第2-99 页的*--pch_messages*, *--no_pch_messages*
- 第2-99 页的*--pch_verbose*, *--no_pch_verbose*
- 第2-125 页的*--use_pch=filename*
- 第4-62 页的*#pragma hdrstop*
- 第4-63 页的*#pragma no_pch*
- *《编译器用户指南》*中第2-18 页的*预编译的头文件*

2.1.35 -D*name*[(*parm-list*)][=*def*]

此选项定义宏 *name*。

语法

-D*name*[(*parm-list*)][=*def*]

其中：

name 是要定义的宏的名称。

parm-list 是以逗号分隔的宏参数的可选列表。通过将宏参数列表追加到宏名称中，可以定义函数风格的宏。

参数列表必须括在括号内。指定多个参数时，不能在列表中的逗号和参数名称之间留有空格。

注意

在 UNIX 系统上，括号可能需要转义。

=*def* 是一个可选的宏定义。

如果省略 =*def*，则编译器会将 *name* 定义为值 1。

若要在命令行中包含识别为标记的字符，请将宏定义括号双引号内。

用法

指定 -D*name* 与将文本 #define *name* 放在每个源文件的开头的作用相同。

限制

编译器按以下顺序定义和取消定义宏：

1. 编译器预定义的宏
2. 使用 -D*name* 显式定义的宏
3. 使用 -U*name* 显式取消定义的宏。

示例

在命令行中指定以下选项：

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

相当于在每个源文件开头定义以下宏：

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

二者效果相同。

另请参阅

- 第2-21 页的*-C*
- 第2-51 页的*-E*
- 第2-123 页的*-Uname*
- 第4-115 页的*编译器预定义*

2.1.36 --data_reorder, --no_data_reorder

此选项允许或禁止对顶级数据项 （如全局变量）进行自动重新排序。

编译器可通过消除数据项之间的多余空格来节省内存。但如果旧代码对有关编译器执行的数据排序的假定无效，则 --data_reorder 可能会破坏该代码。

ISO C 标准不保证数据顺序，因此必须避免编写依赖于任何假定排序的代码。如果需要数据排序，则应将数据项放入结构中。

缺省设置

缺省选项为 --data_reorder。

2.1.37 --debug, --no_debug

此选项允许或禁止在当前编译过程中生成调试表。

无论是否使用 --debug，编译器都会生成相同的代码。唯一差别就是调试表存在与否。

缺省设置

缺省选项为 --no_debug。

使用 --debug 不会影响优化设置。缺省情况下，单独使用 --debug 选项相当于使用以下选项：

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

另请参阅

- *--debug_macros, --no_debug_macros*
- 第2-51 页的*--dwarf2*
- 第2-51 页的*--dwarf3*
- 第2-94 页的*-Onum*

2.1.38 --debug_macros, --no_debug_macros

此选项允许或禁止生成预处理程序宏定义的调试表条目。

用法

使用 --no_debug_macros 可减少调试映像的大小。

此选项必须与 --debug 选项结合使用。

缺省设置

缺省选项为 --debug_macros。

另请参阅

- 第2-36 页的*--debug, --no_debug*

2.1.39 --default_extension=*ext*

此选项使您可以将对象文件的扩展名从缺省扩展名 (.o) 更改为所选扩展名。

语法

--default_extension=*ext*

其中：

ext 是所选的文件扩展名。

示例

以下示例创建一个名为 test.obj 而不是 test.o 的对象文件：

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

注意

-o *filename* 选项可覆盖此文件名。例如，以下命令将产生一个名为 test.o 的对象文件：

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

2.1.40 --dep_name, --no_dep_name

此选项启用或禁用 C++ 中的从属名称处理。

C++ 标准规定应在下列情况下在模板中执行名称查找：

- 在分析模板时 （如果名称不是从属的）
- 在分析模板或实例化模板时 （如果名称是从属的）。

如果选择了选项 --no_dep_name，则只能在实例化模板时在模板中执行从属名称查找。也就是说，不能在分析模板时执行从属名称查找。

注意

--no_dep_name 选项仅作为不符合 C++ 标准的旧式源代码的迁移辅助选项。不建议使用此选项。

模式

仅当源语言为 C++ 时，此选项才有效。

缺省设置

缺省为 --dep_name。

限制

--dep_name 选项不能与 --no_parse_templates 选项结合使用，因为在启用从属名称处理时，缺省情况下将执行分析。

错误

如果将 --dep_name 选项与 --no_parse_templates 结合使用，编译器将会生成错误。

另请参阅

- 第2-97 页的*--parse_templates, --no_parse_templates*
- 第5-14 页的*模板实例化*

2.1.41 --depend=*filename*

此选项指示编译器在编译期间将 makefile 相关性行写入文件中。

语法

--depend=*filename*

其中：

filename 是要输出的相关性文件的名称。

限制

如果在命令行中指定了多个源文件，则将忽略任何 --depend 选项。在这种情况下，不会生成任何相关性文件。

用法

输出文件适合由 make 实用程序使用。若要将输出格式更改为与 UNIX make 实用程序兼容的格式，请使用 --depend_format 选项。

另请参阅

- 第2-40 页的*--depend_format=string*
- 第2-41 页的*--depend_system_headers, --no_depend_system_headers*
- 第2-42 页的*--depend_target=target*
- 第2-70 页的*--ignore_missing_headers*
- 第2-80 页的*--list*
- 第2-86 页的*-M*
- 第2-87 页的*--md*
- 第2-101 页的*--phony_targets*

2.1.42 --depend_format=*string*

此选项更改输出相关性文件的格式，以便与某些 UNIX make 程序兼容。

语法

--depend_format=*string*

其中，*string* 是下列值之一：

unix　使用 UNIX 风格的路径分隔符生成相关性文件条目。

unix_escaped　与 unix 相同，但将空格转义为 \。

unix_quoted　与 unix 相同，但路径名用双引号括起。

用法

unix　在 Windows 系统上，--depend_format=unix 强制使用 UNIX 风格的路径名。也就是说，将使用 UNIX 风格的路径分隔符 / 替代 \。

在 UNIX 系统上，--depend_format=unix 无效。

unix_escaped　在 Windows 系统上，--depend_format=unix_escaped 强制使用 Unix 风格的路径名，并将空格转义为 \。

在 UNIX 系统上，--depend_format=unix_escaped 将空格转义为 \。

unix_quoted　在 Windows 系统上，--depend_format=unix_quoted 强制使用 Unix 风格的路径名，并用 "" 括起这些路径名。

在 UNIX 系统上，--depend_format=unix_quoted 用 "" 括起路径名。

缺省设置

如果未指定 --depend_format 选项，则输出相关性文件的格式将因所选的操作系统而异：

Windows　在 Windows 系统上，缺省情况下使用 Windows 风格的路径或 UNIX 风格的路径 （以指定的为准）。

UNIX　在 UNIX 系统上，缺省为 --depend_format=unix。

示例

在 Windows 系统上，编译包含以下一行的 main.c 文件：

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

上述编译是使用 --depend=depend.txt --depend_format=unix_escaped 选项执行的，这会生成一个包含下列条目的相关性文件 depend.txt：

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

另请参阅

- 第2-39 页的*--depend=filename*
- *--depend_system_headers*, *--no_depend_system_headers*
- 第2-42 页的*--depend_target=target*
- 第2-70 页的*--ignore_missing_headers*
- 第2-86 页的*-M*
- 第2-87 页的*--md*
- 第2-101 页的*--phony_targets*

2.1.43 --depend_system_headers, --no_depend_system_headers

使用 -M 选项或 --md 选项生成 makefile 相关性信息时，此选项用于启用或禁用系统包含相关性行的输出。

缺省设置

缺省选项为 --depend_system_headers。

示例

```
/* hello.c */
#include <stdio.h>
int main(void)
{
    printf("Hello, world!\n");
    return 0;
}
```

用 -M 选项编译此代码时，所生成的结果如下：

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

使用 -M --no_depend_system_headers 选项编译此代码时，所生成的结果如下：

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

另请参阅

- 第2-39 页的*--depend=filename*
- 第2-40 页的*--depend_format=string*
- *--depend_target=target*
- 第2-70 页的*--ignore_missing_headers*
- 第2-86 页的*-M*
- 第2-87 页的*--md*
- 第2-101 页的*--phony_targets*

2.1.44 --depend_target=*target*

此选项为生成 makefile 相关性设置目标。

用法

使用此选项可覆盖缺省目标。

限制

此选项的行为与 GCC 中的 -MT 类似。但是，在指定多个目标时，行为会有所不同。例如， gcc -M -MT target1 -MT target2 file.c 得出的结果可能为 target1 target2: file.c header.h，而 --depend_target=target1 --depend_target=target2 将 target2 视为目标。

另请参阅

- 第2-39 页的*--depend=filename*
- 第2-40 页的*--depend_format=string*
- 第2-41 页的*--depend_system_headers*, *--no_depend_system_headers*
- 第2-70 页的*--ignore_missing_headers*
- 第2-86 页的*-M*
- 第2-87 页的*--md*
- 第2-101 页的*--phony_targets*

2.1.45 --device=list

此选项列出可用于 --device=*name* 选项的受支持的设备名称。

另请参阅

- *--device=name*

2.1.46 --device=*name*

使用此选项可以为特定微控制器或*芯片上系统* (SoC) 设备编译代码。

语法

--device=*name*

其中：

name 是目标微控制器或 SoC 设备的名称。

用法

指定特定设备名称时，设备会从相应的 CPU 继承缺省端标记和浮点体系结构。您可以使用 --bi、--li 和 --fpu 选项来改变端标记和目标浮点体系结构的缺省设置。

另请参阅

- 第2-17 页的*--bigend*
- *--device=list*
- 第2-61 页的*--fpu=name*
- 第2-83 页的*--littleend*
- *《链接器参考指南》*中第2-14 页的*--device=list*
- *《链接器参考指南》*中第2-14 页的*--device=name*
- *《汇编器指南》*中第3-43 页的*使用 C 预处理程序*

2.1.47 --diag_error=*tag*[,*tag*,...]

此选项将具有指定标签的诊断消息设置为 “错误”严重性。

注意

此选项使 #pragma 等效于 #pragma diag_error。

语法

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

其中：

tag[,*tag*,...] 是一个以逗号分隔的诊断消息编号列表，用于指定要更改其严重性的消息。

必须至少指定一个诊断消息编号。

指定多个诊断消息编号时，不能在列表中的逗号和参数名称之间留有空格。

用法

可更改下列类型的诊断消息的严重性：

- 具有编号格式 #*nnnn*-D 的消息。
- 编号格式为 C*nnnn*W 的警告消息。

另请参阅

- 第2-45 页的*--diag_remark=tag[,tag,...]*
- 第2-46 页的*--diag_suppress=tag[,tag,...]*
- 第2-48 页的*--diag_warning=tag[,tag,...]*
- 第4-59 页的*#pragma diag_error tag[,tag,...]*
- *《编译器用户指南》*中第6-5 页的*更改诊断消息的严重性*。

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

此选项将具有指定标记的诊断消息设置为 “备注”严重性。

--diag_remark 选项的行为与 --diag_errors 类似，只不过编译器将具有指定标签的诊断消息设置为 “备注”严重性，而不是 “错误”严重性。

注意

缺省情况下不显示备注。要查看备注消息，请使用编译器选项 --remarks。

注意

此选项使 #pragma 等效于 #pragma diag_remark。

语法

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

其中：

tag[,*tag*,...] 是一个以逗号分隔的诊断消息编号列表，用于指定要更改其严重性的消息。

另请参阅

- 第2-44 页的*--diag_error=tag[,tag,...]*
- 第2-46 页的*--diag_suppress=tag[,tag,...]*
- 第2-48 页的*--diag_warning=tag[,tag,...]*
- 第2-108 页的*--remarks*
- 第4-59 页的*#pragma diag_remark tag[,tag,...]*
- *《编译器用户指南》*中第6-5 页的*更改诊断消息的严重性*。

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

此选项指定用于显示诊断消息的样式。

语法

--diag_style=*string*

其中，*string* 是下列值之一：

arm 使用 ARM 编译器样式显示消息。

ide　　包含任意出错行的行号和字符计数。这些值将显示在括号中。

gnu　　用 gcc 所使用的格式显示消息。

缺省设置

如果未指定 --diag_style 选项，编译器将采用 --diag_style=arm。

用法

选择 --diag_style=ide 选项会隐式选择 --brief_diagnostics 选项。在命令行中显式选择 --no_brief_diagnostics 会覆盖 --diag_style=ide 隐式选择的 --brief_diagnostics。

选择 --diag_style=arm 选项或 --diag_style=gnu 选项不会隐式选择任何 --brief_diagnostics。

另请参阅

- 第2-44 页的*--diag_error=tag[,tag,...]*
- 第2-45 页的*--diag_remark=tag[,tag,...]*
- *--diag_suppress=tag[,tag,...]*
- 第2-48 页的*--diag_warning=tag[,tag,...]*
- *《编译器用户指南》*中第6-5 页的*更改诊断消息的严重性*。

2.1.50 --diag_suppress=*tag*[,*tag*,...]

此选项禁用具有指定标签的诊断消息。

--diag_suppress 选项的行为与 --diag_errors 类似，只不过编译器将禁用具有指定标签的诊断消息，而不是将其设置为 “错误”严重性。

注意

此选项使 #pragma 等效于 #pragma diag_suppress。

语法

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

其中：

tag[,*tag*,...]　　是一个以逗号分隔的诊断消息编号列表，用于指定要禁止的消息。

另请参阅

- 第2-44 页的*--diag_error=tag[,tag,...]*
- 第2-45 页的*--diag_remark=tag[,tag,...]*
- 第2-48 页的*--diag_warning=tag[,tag,...]*
- 第4-60 页的*#pragma diag_suppress tag[,tag,...]*
- *《编译器用户指南》*中第6-6 页的*禁止显示诊断消息*。

2.1.51 --diag_suppress=optimizations

使用此选项将禁止高级优化的诊断消息。

缺省设置

缺省情况下，优化消息具有 “备注”严重性。指定 --diag_suppress=optimizations 会禁止显示优化消息。

注意

使用 --remarks 选项可查看具有 “备注”严重性的优化消息。

用法

编译器在优化级别 -O3 （如循环展开）进行编译时，会执行某些高级向量和标量优化。使用此选项可以禁止与这些高级优化有关的诊断消息。

示例

```
int factorial(int n)
{
    int result=1;
    while (n > 0)
        result *= n--;
    return result;
}
```

用 -O3 -Otime --remarks --diag_suppress=optimizations 选项编译此代码时，将禁止显示优化消息。

另请参阅

- 第2-46 页的*--diag_suppress=tag[,tag,...]*
- *--diag_warning=optimizations*
- 第2-94 页的*-Onum*
- 第2-96 页的*-Otime*
- 第2-108 页的*--remarks*

2.1.52 --diag_warning=*tag*[,*tag*,...]

此选项将具有指定标签的诊断消息设置为 “警告”严重性。

--diag_warning 选项的行为与 --diag_errors 类似，只不过编译器将具有指定标签的诊断消息设置为 “警告”严重性，而不是 “错误”严重性。

注意

此选项使 #pragma 等效于 #pragma diag_warning。

语法

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

其中：

tag[,*tag*,...] 是一个以逗号分隔的诊断消息编号列表，用于指定要更改其严重性的消息。

另请参阅

- 第2-44 页的*--diag_error=tag[,tag,...]*
- 第2-45 页的*--diag_remark=tag[,tag,...]*
- 第2-46 页的*--diag_suppress=tag[,tag,...]*
- 第4-61 页的*#pragma diag_warning tag[, tag, ...]*
- *《编译器用户指南》*中第6-5 页的*更改诊断消息的严重性*

2.1.53 --diag_warning=optimizations

此选项可将高级优化诊断消息设置为 “警告”严重性。

缺省设置

缺省情况下，优化消息具有 “备注”严重性。

用法

编译器在优化级别 -O3 -Otime （如循环展开）进行编译时，会执行某些高级向量和标量优化。使用此选项可显示与这些高级优化相关的诊断消息。

示例

```
int factorial(int n)
{
    int result=1;
    while (n > 0)
        result *= n--;
    return result;
}
```

用 --vectorize --cpu=Cortex-A8 -O3 -Otime --diag_warning=optimizations 选项编译此代码时，会生成优化警告消息。

另请参阅

- 第2-47 页的*--diag_suppress=optimizations*
- 第2-48 页的*--diag_warning=tag[,tag,...]*
- 第2-94 页的*-Onum*
- 第2-96 页的*-Otime*

2.1.54 --dllexport_all, --no_dllexport_all

此选项使您可以控制生成 DLL 时的符号可见性。

缺省设置

缺省为 --no_dllexport_all。

用法

使用 --dllexport_all 选项可将所有 extern 定义标记为 __declspec(dllexport)。

另请参阅

- 第2-4 页的*--apcs=qualifer...qualifier*
- 第4-24 页的*__declspec(dllexport)*

2.1.55 --dllimport_runtime, --no_dllimport_runtime

此选项用于控制在运行时库用作共享库时的符号可见性。

缺省设置

缺省为 --no_dllimport_runtime。

用法

使用 --dllimport_runtime 选项可以：

- 将所有内置符号标记为 __declspec(dllimport)
- 标记 cpprt 运行时库中生成的*运行时类型信息* (RTTI) 以便执行导入
- 在将原函数标记为 __declspec(dllimport) 的前提下，标记任何优化的 printf() 和 __hardfp_ 函数以便执行导入。

另请参阅

- 第2-68 页的*--guiding_decls, --no_guiding_decls*
- 第2-110 页的*--rtti, --no_rtti*
- 第4-26 页的*__declspec(dllimport)*

2.1.56 --dollar, --no_dollar

此选项指示编译器在标识符中接受或拒绝美元符号 $。

缺省设置

如果未指定 --strict 选项，则缺省为 --dollar。

如果指定了 --strict 选项，则缺省为 --no_dollar。

另请参阅

- 第3-12 页的*标识符中的美元符号*
- 第2-115 页的*--strict, --no_strict*

2.1.57 --dwarf2

此选项指示编译器使用 DWARF 2 调试表格式。

缺省设置

除非显式指定 --dwarf2，否则编译器将采用 --dwarf3。

另请参阅

- *--dwarf3*

2.1.58 --dwarf3

此选项指示编译器使用 DWARF 3 调试表格式。

缺省设置

除非显式指定 --dwarf2，否则编译器将采用 --dwarf3。

另请参阅

- *--dwarf2*

2.1.59 -E

此选项指示编译器仅执行预处理程序步骤。

缺省情况下，预处理程序的输出将发送到标准输出流，并且可以用标准 UNIX 和 MS-DOS 记号重定向到文件。

也可以使用 -o 选项指定用于存放预处理输出的文件。缺省情况下，将从输出中删除注释。预处理程序接受带有任何扩展名 （如 .o、 .s 和 .txt）的源文件。

示例

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

另请参阅

- 第2-21 页的*-C*
- 第2-92 页的*-o filename*

2.1.60 --emit_frame_directives, --no_emit_frame_directives

此选项指示编译器将 DWARF FRAME 指令放置在反汇编输出中。

缺省设置

缺省选项为 --no_emit_frame_directives。

示例

```
armcc --asm --emit_frame_directives foo.c

armcc -S emit_frame_directives foo.c
```

另请参阅

- 第2-15 页的*--asm*
- 第2-110 页的*-S*
- *《汇编器指南》*中第2-50 页的*使用框架指令*。

2.1.61 --enum_is_int

此选项将所有枚举类型的大小强制设置为至少四字节。

缺省情况下禁用此选项，并使用可存放所有枚举器的值的最小数据类型。

如果在命令行中指定 ARM Linux 配置文件，缺省情况下会启用此选项。

——注意——

建议尽量少用 --enum_is_int 选项。

另请参阅

- 第2-10 页的*--arm_linux_config_file=path*
- 第2-11 页的*--arm_linux_configure*
- 第2-73 页的*--interface_enums_are_32_bit*

2.1.62 --errors=*filename*

此选项将诊断消息的输出从 stderr 重定向到指定的错误文件。

语法

--errors=*filename*

其中：

filename 是要将错误重定向到的文件的名称。

在某些情况下 （例如，错误地键入了选项名称），不会重定向与命令选项问题相关的诊断。不过，如果为选项指定了无效参数 （如 --cpu=999），就会将相关的诊断重定向到指定的 *filename*。

另请参阅

- 第2-19 页的*--brief_diagnostics, --no_brief_diagnostics*
- 第2-44 页的*--diag_error=tag[,tag,...]*
- 第2-45 页的*--diag_remark=tag[,tag,...]*
- 第2-45 页的*--diag_style={arm|ide|gnu}*
- 第2-46 页的*--diag_suppress=tag[,tag,...]*
- 第2-48 页的*--diag_warning=tag[,tag,...]*
- 第2-108 页的*--remarks*
- 第2-129 页的*-W*
- 第2-133 页的*--wrap_diagnostics, --no_wrap_diagnostics*
- *《编译器用户指南》*中的第 6 章 *诊断消息*。

2.1.63 --exceptions, --no_exceptions

此选项启用或禁用异常处理。

在 C++ 中， --exceptions 选项允许使用 throw 和 try/catch，以便遵守函数异常规范，并使编译器发布展开表以支持运行时的异常传播。

在 C++ 中，如果指定了 --no_exceptions 选项，则不允许在源代码中使用 throw 和 try/catch。但是，仍会分析函数异常规范，不过将忽略其大部分意义。

在 C 中，如果通过已编译的函数抛出了异常，则用 --no_exceptions 编译的代码行为将被取消定义。如果希望通过 C 函数正确传播异常，必须使用 --exceptions。

缺省设置

缺省为 --no_exceptions。但是，如果在命令行中指定 ARM Linux 配置文件并使用 --translate_g++，缺省值将更改为 --exceptions。

另请参阅

- 第2-10 页的*--arm_linux_config_file=path*
- 第2-11 页的*--arm_linux_configure*
- *--exceptions_unwind, --no_exceptions_unwind*

2.1.64 --exceptions_unwind, --no_exceptions_unwind

此选项启用或禁用针对识别异常的代码来展开函数。仅当启用 --exceptions 时，此选项才有效。

如果使用 --no_exceptions_unwind 和 --exceptions，则不能通过已编译函数传播异常，而应调用 std::terminate。

缺省设置

缺省选项为 --exceptions_unwind。

另请参阅

- 第2-53 页的*--exceptions, --no_exceptions*
- 第5-17 页的*运行时的函数展开*

2.1.65 --export_all_vtbl, --no_export_all_vtbl

此选项控制在 C++ 中导出动态符号的方式。

模式

仅当源语言为 C++ 时，此选项才有效。

缺省设置

缺省为 --no_export_all_vtbl。

用法

使用选项 --export_all_vtbl 可导出具有关键函数的类的所有虚拟函数表和 RTTI。*关键函数* 是类中按非内联的声明顺序列出的第一个虚拟函数，它不是纯虚拟函数。

注意

可以通过使用 __declspec(notshared) 来禁用特定类的导出。

另请参阅

- 第4-29 页的*__declspec(notshared)*

2.1.66 --export_defs_implicitly, --no_export_defs_implicitly

此选项控制导出动态符号的方式。

缺省设置

缺省选项为 --no_export_defs_implicitly。

用法

使用 --export_defs_implicitly 选项可以导出其原型标记为 __declspec(dllimport) 的定义。

另请参阅

- 第4-26 页的*__declspec(dllimport)*

2.1.67 --extended_initializers, --no_extended_initializers

这些选项启用和禁用使用扩展常数初始值设定项，甚至使用 --strict 或 --strict_warnings 进行编译也是如此。

当使用某些不可移植但受到广泛支持的常数初始值设定项 （如地址到整数类型的类型转换）时，--extended_initializers 使编译器生成相同的常规警告 （与通常在非严格模式下生成的常数初始值设定项有关），而不是生成特定错误来指出表达式必须具有常数值或算术类型。

缺省设置

如果用 --strict 或 --strict_warnings 进行编译，则缺省选项为 --no_extended_initializers。

如果在非严格模式下进行编译，则缺省选项为 --extended_initializers。

另请参阅

- 第2-115 页的*--strict, --no_strict*
- 第2-116 页的*--strict_warnings*
- 第3-9 页的*常数表达式*

2.1.68 --feedback=*filename*

此选项可有效消除未使用的函数，在 ARMv4T 体系结构上，可以减少交互操作所需的编译。

语法

--feedback=*filename*

其中：

filename 是由 ARM 链接器在上次执行时创建的反馈文件。

用法

可以使用同一反馈文件执行多次编译。编译器将反馈文件中标识的每个未使用函数放入对应对象文件中其自身的 ELF 节中。

反馈文件包含上次编译的相关信息。这是因为：

- 反馈文件可能已过期。也就是说，当前源代码中可能使用了以前标识为未使用的函数。仅当未使用函数也未用于当前源代码时，链接器才会删除其代码。

  注意

  — 因此，使用链接器反馈消除未使用函数是安全的优化，但对代码大小的影响不大。

  — 对于交互操作而言，缩减编译的使用要求比消除未使用函数要严格的多。如果要缩减交互操作编译，一定要确保反馈文件以及生成该文件的源代码保持为最新，这一点非常重要。

- 若要从链接器反馈获得最大益处，必须至少执行两次完全编译和链接。不过，通常使用上次编译中的反馈执行单次编译和链接就足够了。

另请参阅

- 第2-114 页的*--split_sections*
- *《链接器参考指南》*中第2-22 页的*--feedback_type=type*
- *《编译器用户指南》*中第2-26 页的*使用链接器反馈*。

2.1.69 --force_new_nothrow, --no_force_new_nothrow

此选项控制 C++ 中 new 表达式的行为。

C++ 标准规定只允许用 throw() 声明的无抛出 operator new 在失败时返回 NULL。从不允许任何其他 operator new 在失败时返回 NULL，并且缺省的 operator new 在失败时将抛出异常。

如果使用 --force_new_nothrow，则编译器会将使用全局 ::operator new 或 ::operator new[] 的表达式 （如 new T(...args...)）视为 new (std::nothrow) T(...args...)。

此外，--force_new_nothrow 还会使编译器将类特定的任何 operator new 或任何重载的全局 operator new 视为无抛出。

注意

--force_new_nothrow 选项仅作为不符合 C++ 标准的旧式源代码的迁移辅助选项。不建议使用此选项。

模式

仅当源语言为 C++ 时，此选项才有效。

缺省设置

缺省为 --no_force_new_nothrow。

示例

```
struct S
{
    void* operator new(std::size_t);
```

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

如果启用 --force_new_nothrow 选项，则上述示例将被视为：

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

另请参阅

- 第5-13 页的*使用 ::operator new 函数*

2.1.70 --forceinline

此选项强制所有内联函数被视为用 __forceinline 进行限定。

内联函数是用 inline 或 __inline 限定的函数。在 C++ 中，内联函数是在结构、类或联合定义内部定义的函数。

如果使用 --forceinline，编译器总是尽量尝试内联这些函数。但如果对函数进行内联时出现问题，则编译器将不再内联。例如，递归函数仅内联到本身一次。

另请参阅

- 第2-16 页的*--autoinline, --no_autoinline*
- 第2-73 页的*--inline, --no_inline*
- 第4-6 页的*__forceinline*
- 第4-9 页的*__inline*
- *《编译器用户指南》*中第5-17 页的*管理内联*。

2.1.71 --fp16_format=*format*

此选项将半精度浮点数作为 VFPv3 体系结构的可选扩展。如果未指定格式，使用 __fp16 数据类型会被编译器视为错误。

语法

--fp16_format=*format*

其中， *format* 是下列值之一：

alternative　ieee 的替代值，可提供附加范围，但不含非数字或无穷大值。

ieee　IEEE 754r 定义的半精度二进制浮点格式， IEEE 754r 是 IEEE 754 标准的修订版。

none　这是缺省设置。这相当于未指定格式，意味着编译器会将使用 __fp16 数据类型视为错误。

另请参阅

- 第E-4 页的*内在函数*
- *《编译器用户指南》*中第5-33 页的*半精度浮点数支持*。

2.1.72 --fpmode=*model*

此选项指定浮点遵从性，并设置库属性和浮点优化。

语法

--fpmode=*model*

其中， *model* 是下列值之一：

ieee_full　由 IEEE 标准保证的所有工具、操作和表示对单精度和双精度都可用。可在运行时动态选择操作模式。

此选项定义下列符号：

__FP_IEEE
__FP_FENV_EXCEPTIONS
__FP_FENV_ROUNDING
__FP_INEXACT_EXCEPTION

ieee_fixed　使用舍入到最接近的数且无不精确异常的 IEEE 标准。

此选项定义下列符号：

```
__FP_IEEE
__FP_FENV_EXCEPTIONS
```

ieee_no_fenv

使用舍入到最接近的数且无异常的 IEEE 标准。此模式是无状态的，并与 Java 浮点算术模型兼容。

此选项定义符号 __FP_IEEE。

std　使用非正规数清零、舍入到最接近的数且无异常的 IEEE 有限值。此选项与标准 C 和 C++ 兼容，并且是缺省选项。

将根据 IEEE 标准预测标准有限值。但存在下列限制：

- 可能无法在 IEEE 模型定义的所有环境中生成非数字和无穷大。即使生成了，它们的符号也可能有所不同。
- 零符号可能不是 IEEE 模型预测的符号。

fast　执行会对精度产生较小损失的更积极的浮点优化，以显著提高性能。此选项定义符号 __FP_FAST。

此选项会导致与 ISO C 或 C++ 标准不完全兼容的行为。但数字稳健的浮点程序的行为应该是正确的。

可执行许多转换，其中包括：

- 如果所有浮点参数都可完全以单精度值形式表示，则双精度数学函数可转换为等效的单精度函数，并且会直接将结果转换为单精度值。

  仅当所选库包含等效的单精度函数（例如，所选库为 rvct 或 aeabi_glibc）时，才会执行此转换。

  例如：

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

  将转换为

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

- 对于限定为单精度的双精度浮点表达式，如果比较有利，会用单精度进行计算。例如，`float y = (float)(x + 1.0)` 将以 `float y = (float)x + 1.0f` 形式计算。
- 将除以浮点常数替换为乘以其倒数。例如，`x / 3.0` 将以 `x * (1.0 / 3.0)` 形式计算。
- 在调用数学函数后，不能保证 `errno` 的值与 ISO C 或 C++ 标准兼容。这使编译器可以对 VFP 平方根指令进行内联，而不是调用 `sqrt()` 或 `sqrtf()`。

注意

可能需要初始化代码来启用 VFP。有关详细信息，请参阅 *《编译器用户指南》*中第5-32 页的*VFP 支持*。

另请参阅

- *install_directory*\RVDS\Examples\...\vfpsupport 中的 *《ARM 应用程序注释 133 - 在 RVDS 中使用 VFP》*

2.1.73 --fpu=list

此选项列出可与 `--fpu=name` 选项结合使用的受支持的 FPU 体系结构名称。

未列出已被弃用的选项。

另请参阅

- *--fpu=name*

2.1.74 --fpu=*name*

此选项使您可以确定目标 FPU 体系结构。

如果指定此选项，则在某些情况下 （例如，使用 `--cpu` 选项时），它将覆盖显示在命令行中的所有隐式 FPU 选项。

若要获取 FPU 体系结构的完整列表，请使用 `--fpu=list` 选项。

语法

`--fpu=`*name*

其中， *name* 是下列值之一：

none　选择无浮点选项。这样不会使用浮点代码。如果代码包含 **float** 类型，则会生成错误。

vfpv　这是 vfpv2 的同义词。

vfpv2　选择符合 VFPv2 体系结构的硬件向量浮点单元。

注意

如果在命令行中输入 armcc --thumb --fpu=vfpv2，则编译器会尽可能使用 Thumb 指令集编译代码，但会将与硬件浮点有关的函数编译为 ARM 代码。在这种情况下，预定义 __thumb 的值是不正确的。

如果用 --arm 选项为 ARM C 代码指定了 vfp 或 vfpv2，则必须使用 __softfp 关键字确保将交互操作的 ARM 代码编译为使用软件浮点链接。

vfpv3　选择符合 VFPv3 体系结构的硬件向量浮点单元。 VFPv3 能向后兼容 VFPv2，但 VFPv3 不能捕获浮点异常。只有 RealView Development Suite 3.0 及更高版本才提供 vpfv3。

vfpv3_fp16　选择符合 VFPv3 体系结构的硬件向量浮点单元，该体系结构也提供半精度扩展。只有 RealView Development Suite 4.0 及更高版本才提供 vfpv3_fp16。

vfpv3_d16　选择符合 VFPv3-D16 体系结构的硬件向量浮点单元。只有 RealView Development Suite 4.0 及更高版本才提供 vfpv3_d16。

vfpv3_d16_fp16

选择符合 VFPv3-D16 体系结构的硬件向量浮点单元，该体系结构也提供半精度扩展。只有 RealView Development Suite 4.0 及更高版本才提供 vfpv3_d16_fp16。

softvfp　选择软件浮点库 fplib。如果未指定 --fpu 选项或者选择了不带 FPU 的 CPU，则此选项为缺省选项。

在早期版本的 RVCT 中，如果指定了 --fpu=softvfp 和具有隐式 VFP 硬件的 CPU，则链接器会选择使用 VFP 指令实现软浮点调用的库。但现在此行为已不再有效。如果需要这一旧式行为，请使用 --fpu=softvfp+vfp。

softvfp+vfpv2

选择浮点库，该库具有可以使用 VFPv2 指令的软件浮点链接。若要在实现 VFP 单元的系统上对 Thumb 代码与 ARM 代码进行交互操作，请选择此选项。

如果选择此选项，则：

- 用 --thumb 进行编译的行为与 --fpu=softvfp 类似，只不过前者与使用 VFP 指令的浮点库相链接。
- 用 --arm 编译的行为与 --fpu=vfpv2 类似，只不过前者为所有函数指定了软浮点链接。这意味着函数传递及返回浮点参数和结果的方式与 --fpu=softvfp 相同，只不过前者是在内部使用 VFP 指令。

注意

如果用 --arm 或 --thumb 选项为 C 代码指定了 softvfp+vfpv2，则应确保将交互操作的浮点代码编译为使用软浮点链接。

softvfp+vfpv3

选择浮点库，该库具有以 VFPv3 体系结构为目标的软件浮点链接。若要在实现 VFPv3 单元的系统上对 Thumb 代码与 ARM 代码进行交互操作，请选择此选项。只有 RealView Development Suite 3.0 及更高版本才提供 softvfp+vfpv3。

softvfp+vfpv3_fp16

选择浮点库，该库具有以支持半精度浮点扩展的 VFPv3 体系结构为目标的软件浮点链接。只有 RealView Development Suite 4.0 及更高版本才提供 softvfp+vfpv3_fp16。

softvfp+vfpv3_d16

选择浮点库，该库具有以 VFPv3-D16 体系结构为目标的软件浮点链接。只有 RealView Development Suite 4.0 及更高版本才提供 softvfp+vfpv3_d16。

softvfp+vfpv3_d16_fp16

选择浮点库，该库具有以支持半精度浮点扩展的 VFPv3-D16 体系结构为目标的软件浮点链接。只有 RealView Development Suite 4.0 及更高版本才提供 softvfp+vfpv3_d16_fp16。

限制

使用 --fpu 选项显式选择的任何 FPU 始终覆盖使用 --cpu 选项隐式选定的任何 FPU。例如，即使对 CPU 的选择隐含了使用体系结构 VFPv2，选项 --cpu=ARM1136JF-S --fpu=softvfp 也将生成使用软件浮点库 fplib 的代码。

如果使用 --cpu 选项隐式指定的 FPU 与使用 --fpu 显式选择的 FPU 不兼容，则编译器将生成错误。

编译器只生成标量浮点运算。如果要使用 VFP 向量运算，必须使用汇编代码来实现。

已对 softvfp 禁用 NEON 支持。

缺省设置

如果存在 VFP 协处理器，则会生成 VFP 指令。如果不存在 VFP 协处理器，编译器生成调用软件浮点库 fplib 的代码，用于执行浮点运算。

注意

缺省情况下，选择某些处理器或体系结构会隐式选择特定浮点单元。例如，选项 --cpu ARM1136JF-S 隐含了选项 --fpu vfpv2。

另请参阅

- 第2-8 页的*--arm*
- 第2-30 页的*--cpu=name*
- 第2-118 页的*--thumb*
- 第4-15 页的*__softfp*
- *《编译器用户指南》*中第5-35 页的*浮点计算和链接*
- *《开发指南》*中第2-5 页的*浮点生成选项*。

2.1.75 --friend_injection, --no_friend_injection

此选项控制 C++ 中 friend 声明的可见性。

在 C++ 中，此选项控制使用标准查找机制时，仅在 friend 声明中声明的类或函数的名称是否可见。

声明 friend 名称后，它们对这些查找是可见的。如果未按标准要求声明 friend 名称，则仅当使用参数相关查找并且类名从不可见时，函数名称才可见。

注意

--friend_injection 选项仅作为不符合 C++ 标准的旧式源代码的迁移辅助选项。不建议使用此选项。

模式

仅当源语言为 C++ 时，此选项才有效。

缺省设置

缺省选项为 --no_friend_injection。

另请参阅

- 第3-14 页的*friend*

2.1.76 -g

此选项用于在当前编译过程中生成调试表。

无论是否使用 -g，编译器都会生成相同的代码。唯一差别就是调试表存在与否。

使用 -g 不会影响优化设置。缺省情况下，单独使用 -g 选项相当于使用以下选项：

```
-g --dwarf3 --debug_macros
```

另请参阅

- 第2-36 页的*--debug, --no_debug*
- 第2-37 页的*--debug_macros, --no_debug_macros*
- 第2-51 页的*--dwarf2*
- 第2-51 页的*--dwarf3*
- 第2-94 页的*-Onum*

2.1.77 --global_reg=*reg_name*[,*reg_name*,...]

此选项将指定的寄存器名称视为固定寄存器。

语法

--global_reg=*reg_name*[,*reg_name*,...]

其中， *reg_name* 是寄存器的 APCS 或 TPCS 名称，由 1 到 8 的整数值表示。

寄存器名称 1 到 8 按顺序映射到寄存器 r4 到 r11。

限制

此选项与 __global_reg 存储类说明符具有相同的限制。

示例

--global_reg=1,4,5 // 分别保留寄存器 r4、 r7 和 r8。

另请参阅

- 第4-7 页的*__global_reg*
- 《ARM 软件开发工具包参考指南》 (*ARM Software Development Toolkit Reference Guide*)。

2.1.78 --gnu

此选项启用 ARM 编译器支持的 GNU 编译器扩展。通过检查预定义宏 __GNUC__ 和 __GNUC_MINOR__ 可以确定这些扩展所兼容的 GCC 版本。

另请参阅

- 第2-22 页的*--c90*
- 第2-22 页的*--c99*
- 第2-30 页的*--cpp*
- 第2-115 页的*--strict*, *--no_strict*
- 第3-23 页的*GNU 语言扩展*
- 第4-115 页的*编译器预定义*

2.1.79 --gnu_instrument, --no_gnu_instrument

此选项插入 GCC 样式的检查。

另请参阅

- *--gnu_instrument*, *--no_gnu_instrument*

2.1.80 --gnu_version=*version*

此选项尝试使编译器与特定版本的 GCC 兼容。

语法

--gnu_version=*version*

其中 *version* 是十进制数，表示尝试使编译器兼容的 GCC 版本。

模式

此选项在使用 GNU 兼容模式时才适用。

用法

此选项供专业人员使用。它用于处理旧代码。您通常无需使用此选项。

缺省设置

在 RVCT v4.0 中，缺省值为 40200。这对应于 GCC 4.2.0 版。

示例

--gnu_version=30401 使编译器尽量与 GCC 3.4.1 兼容。

另请参阅

- 第2-66 页的*--gnu*

2.1.81 --guiding_decls, --no_guiding_decls

此选项为 C++ 中的模板函数启用或禁用定向声明的识别功能。

*定向声明*是一种函数声明，它与函数模板实例相匹配，但因其定义从函数模板派生而没有显式定义。

如果将 --no_guiding_decls 与 --old_specializations 结合使用，则无法识别非成员模板函数的特化。它将被视为独立函数的定义。

注意

--guiding_decls 选项仅作为不符合 C++ 标准的旧式源代码的迁移辅助选项。不建议使用此选项。

模式

仅当源语言为 C++ 时，此选项才有效。

缺省设置

缺省选项为 --no_guiding_decls。

示例

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

如果被视为定向声明，则 f(int) 是模板实例。否则，它将是独立函数，因此必须为其提供定义。

另请参阅

- 第2-4 页的*--apcs=qualifer...qualifier*
- 第2-95 页的*--old_specializations*, *--no_old_specializations*

2.1.82 --help

此选项汇总显示主要的命令行选项。

如果未指定任何选项或源文件，则它是缺省选项。

另请参阅

- 第2-112 页的*--show_cmdline*
- 第2-129 页的*--vsn*

2.1.83 --hide_all, --no_hide_all

此选项用于控制生成 SVr4 共享对象时的符号可见性。

用法

使用 --no_hide_all 可强制编译器对所有不使用 __declspec(dll*) 的 extern 变量和函数使用 STV_DEFAULT 可见性。这也会强制这些变量和函数在运行时通过动态装入程序成为可预占的。

在生成 System V 或 ARM Linux 共享库时，请结合使用 --no_hide_all 和 --apcs /fpic。

缺省设置

缺省为 --hide_all。

另请参阅

- 第2-4 页的*--apcs=qualifer...qualifier*
- 第4-24 页的*__declspec(dllexport)*
- 第4-26 页的*__declspec(dllimport)*
- *《链接器参考指南》*中第4-4 页的*符号可见性*
- *《链接器参考指南》*中第2-57 页的*--symver_script=file*

2.1.84 -I*dir*[,*dir*,...]

此选项将指定的目录或以逗号分隔的目录列表添加到用于查找所包含文件的搜索位置列表。

如果指定了多个目录，则按 -I 选项指定它们的相同顺序搜索这些目录。

语法

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

其中：

dir[,*dir*,...] 是要搜索所包含文件的以逗号分隔的目录列表。

必须至少指定一个目录。

指定多个目录时，不能在列表中的逗号和目录名称之间留有空格。

另请参阅

- 第2-76 页的*-Jdir[,dir,...]*
- 第2-76 页的*--kandr_include*
- 第2-102 页的*--preinclude=filename*
- 第2-117 页的*--sys_include*
- *《编译器用户指南》*中第2-15 页的*头文件*

2.1.85 --ignore_missing_headers

此选项指示编译器输出头文件的相关性行，即使缺少头文件也是如此。

禁止对缺少头文件显示警告消息和错误消息，在此情况下，编译将继续，而在其他情况下将失败。

用法

此选项用于自动更新 makefile。它与 GCC `-MG` 命令行选项类似。

另请参阅

- 第2-39 页的*--depend=filename*
- 第2-40 页的*--depend_format=string*
- 第2-41 页的*--depend_system_headers*, *--no_depend_system_headers*
- 第2-42 页的*--depend_target=target*
- 第2-86 页的*-M*
- 第2-87 页的*--md*
- 第2-101 页的*--phony_targets*

2.1.86 --implicit_include, --no_implicit_include

此选项控制将源文件的隐式包含用作查找要在 C++ 中实例化的模板实体定义的方法。

模式

仅当源语言为 C++ 时，此选项才有效。

缺省设置

缺省选项为 --implicit_include。

另请参阅

- *--implicit_include_searches, --no_implicit_include_searches*
- 第5-15 页的*隐式包含*

2.1.87 --implicit_include_searches, --no_implicit_include_searches

此选项控制编译器搜索 C++ 模板中隐式包含文件的方式。

如果选择 --implicit_include_searches 选项，则编译器将根据 *filename*.* 形式的部分名称使用搜索路径查找隐式包含文件。搜索路径由 -I、-J 和 RVCT40INC 环境变量确定。

如果选择 --no_implicit_include_searches 选项，则编译器将根据文件的完整名称（包括路径名）查找隐式包含文件。

模式

仅当源语言为 C++ 时，此选项才有效。

缺省设置

缺省选项为 --no_implicit_include_searches。

另请参阅

- 第2-69 页的*-Idir[,dir,...]*
- 第2-71 页的*--implicit_include*, *--no_implicit_include*
- 第2-76 页的*-Jdir[,dir,...]*
- 第5-15 页的*隐式包含*
- *《编译器用户指南》*中第2-16 页的*搜索路径*。

2.1.88 --implicit_typename, --no_implicit_typename

此选项控制根据上下文来隐式确定 C++ 中的模板参数从属名称是类型还是非类型。

注意

--implicit_typename 选项仅作为不符合 C++ 标准的旧式源代码的迁移辅助选项。不建议使用此选项。

模式

仅当源语言为 C++ 时，此选项才有效。

缺省设置

缺省为 --no_implicit_typename。

注意

除非指定 --no_parse_templates，否则 --implicit_typename 选项将无效。

另请参阅

- 第2-38 页的*--dep_name*, *--no_dep_name*
- 第2-97 页的*--parse_templates*, *--no_parse_templates*
- 第5-14 页的*模板实例化*

2.1.89 --info=totals

此选项指示编译器为每个对象文件提供对象代码和数据大小的总和。

如果以类似格式使用 fromelf -z，则编译器会返回与 fromelf 相同的总和。当源代码中存在嵌入式汇编代码时，这些总和将包括嵌入式汇编器大小。

示例

```
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
```

(inc. data) 列提供了作为代码组成部分的常数、字符串及其他数据项的大小。此示例中所示的 Code 列*包括*此值。

另请参阅

- 第2-80 页的*--list*
- *《链接器参考指南》*中第2-25 页的*--info=topic[,topic,...]*
- *《编译器用户指南》*中第5-9 页的*代码度量*
- *《实用程序指南》*中第2-2 页的*使用命令行选项*。

2.1.90 --inline, --no_inline

此选项启用或禁用函数的内联。禁用函数内联有助于增强调试模拟效果。

如果选择了 --inline 选项，则编译器会考虑为每个函数进行内联。用 --inline 编译代码并不会保证内联所有函数。有关编译器如何确定对函数进行内联的详细信息，请参阅 *《编译器用户指南》*中第5-17 页的*什么时候才适合编译器执行内联？*。

如果选择了 --no_inline 选项，则编译器不会尝试内联用 __forceinline 限定的函数以外的函数。

缺省设置

缺省选项为 --inline。

另请参阅

- 第2-16 页的*--autoinline, --no_autoinline*
- 第2-58 页的*--forceinline*
- 第2-94 页的*-Onum*
- 第2-96 页的*-Ospace*
- 第2-96 页的*-Otime*
- 第4-6 页的*__forceinline*
- 第4-9 页的*__inline*
- *《编译器用户指南》*中第2-26 页的*使用链接器反馈*
- *《编译器用户指南》*中第5-16 页的*函数内联*。

2.1.91 --interface_enums_are_32_bit

此选项有助于在外部代码接口间提供兼容性，这与枚举类型的大小有关。

用法

无法将使用 --enum_is_int 编译的对象文件与另一个不使用 --enum_is_int 编译的对象文件进行链接。链接器无法确定是否以影响外部接口的方式使用了枚举类型，因此在检测这些生成差异时，它将会生成警告或错误信息。使用 --interface_enums_are_32_bit 进行编译可避免此情况。这样，可将结果对象文件与任何其他对象文件进行链接，而链接器不会检测到因不同枚举类型大小引起的冲突。

——注意——

如果使用此选项，即表示您向编译器保证外部接口中使用的所有枚举类型均为 32 位宽。例如，如果确保声明的每个 enum 都包含至少一个大于 2 到 16 的幂的值，则不论是否使用 --enum_is_int，都会强制编译器令 enum 为 32 位宽。向编译器提供的保证是否真实完全取决于您。（另一种满足此条件的方法是确保外部接口中没有 enums。）

另请参阅

- 第2-52 页的*--enum_is_int*

2.1.92 --interleave

此选项以注释形式逐行交叉存取使用 --asm 或 -S 选项所生成的汇编列表中的 C 或 C++ 源代码。

用法

--interleave 的操作取决于所使用的选项组合：

表 2-4 用 ---interleave 选项进行编译

编译器选项	操作
--asm --interleave	将汇编列表写入已编译源代码的反汇编文件中，从而使源代码与反汇编代码交叉存取。 此外，还会执行链接步骤，除非使用了 -c 选项。 反汇编代码将写入一个文本文件中，其缺省名称为具有文件扩展名 .txt 的输入文件的名称
-S --interleave	将汇编列表写入已编译源代码的反汇编文件中，从而使源代码与反汇编代码交叉存取。 反汇编代码将写入一个文本文件中，其缺省名称为具有文件扩展名 .txt 的输入文件的名称

限制

- 不能重新汇编用 --asm --interleave 或 -S --interleave 生成的汇编列表。
- 预处理源文件包含 #line 指令。在使用 --asm --interleave 或 -S --interleave 编译预处理文件时，编译器将搜索任意 #line 指令所指示的原始文件，并使用这些文件中适当的行。这样可以确保编译预处理文件与编译原始文件的输出和行为完全相同。

  如果编译器找不到原始文件，就无法交叉存取源。因此，如果用 #line 指令预处理了源文件，但未经预处理的源文件不存在，则必须先删除所有 #line 指令，然后再用 --interleave 进行编译。

另请参阅

- 第2-15 页的*--asm*
- 第2-110 页的*-S*

2.1.93 -J*dir*[,*dir*,...]

此选项将指定的目录或以逗号分隔的目录列表添加到系统包含列表中。

即便使用 --diag_error，也会禁止显示警告和备注。

RVCT40INC 环境变量将设置为缺省系统包含路径，除非使用 -J 覆盖了该变量。在系统包含列表中先搜索带尖括号的包含文件，然后再搜索用 -I 选项指定的任何包含列表。

注意

在 Windows 系统上，如果在命令行中指定了 RVCT40INC，则必须用双引号括起此环境变量，这是因为该变量所定义的缺省路径会包含空格。例如：

```
armcc -J"%RVCT40INC%" -c main.c
```

语法

-J*dir*[,*dir*,...]

其中：

dir[,*dir*,...] 是要添加到系统包含列表中的以逗号分隔的目录列表。

必须至少指定一个目录。

指定多个目录时，不能在列表中的逗号和目录名称之间留有空格。

另请参阅

- 第2-69 页的*-Idir[,dir,...]*
- *--kandr_include*
- 第2-102 页的*--preinclude=filename*
- 第2-117 页的*--sys_include*
- *《编译器用户指南》*中第2-15 页的*头文件*

2.1.94 --kandr_include

此选项确保使用 Kernighan 和 Ritchie 搜索规则查找所包含的文件。

当前位置由初始源文件定义并且不进入堆栈。如果未使用此选项，则将使用 Berkeley 式搜索。

另请参阅

- 第2-69 页的*-Idir[,dir,...]*
- 第2-76 页的*-Jdir[,dir,...]*
- 第2-102 页的*--preinclude=filename*
- 第2-117 页的*--sys_include*
- *《编译器用户指南》*中第2-15 页的*头文件*
- *《编译器用户指南》*中第2-15 页的*当前位置*。

2.1.95 -L*opt*

此选项指定在编译后执行链接步骤时要传递给链接器的命令行选项。可在创建部分链接的对象或可执行映像时传递命令行选项。

语法

-L*opt*

其中：

opt 是要传递给链接器的命令行选项。

限制

如果使用 -L 向链接器传递了不受支持的链接器选项，则会生成错误。

示例

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

另请参阅

- 第2-2 页的*-Aopt*
- 第2-112 页的*--show_cmdline*

2.1.96 --library_interface=*lib*

此选项启用与所选库类型兼容的代码的生成。

语法

--library_interface=*lib*

其中，*lib* 是下列值之一：

rvct
: 指定编译器输出使用 RVCT 运行时库。

rvct_c90
: 其行为与 --library_interface=*rvct* 类似。区别在于编译器不修改输入源代码中引用的未由 C90 保留的函数名。否则，某些 C99 math.h 函数名可能以 __hardfp_ 为前缀，例如 __hardfp_tgamma。

aeabi_clib90
: 指定编译器输出使用符合 *ARM 嵌入式应用程序二进制接口* (AEABI) 的任何 ISO C90 库。

aeabi_clib99
: 指定编译器输出使用符合 *ARM 嵌入式应用程序二进制接口* (AEABI) 的任何 ISO C99 库。

aeabi_clib
: 指定编译器输出使用符合 *ARM 嵌入式应用程序二进制接口* (AEABI) 的任何 ISO C 库。

  选择 --library_interface=aeabi_clib 选项相当于指定 --library_interface=aeabi_clib90 或 --library_interface=aeabi_clib99，具体取决于所使用的源语言选项。

  源语言的选项将因所选的命令行选项以及所使用的文件名后缀而异。

aeabi_glibc
: 指定编译器输出使用符合 AEABI 版本的 GNU C 库。

缺省设置

如果未指定 --library_interface，编译器将采用 --library_interface=rvct。

用法

- 使用 --library_interface=rvct 选项可在链接时利用全套的编译器和库优化。
- 在链接到符合 ABI 的 C 库时，使用 --library_interface=aeabi_* 形式的选项。--library_interface=aeabi_* 形式的选项可确保编译器不会生成对 RVCT C 库所提供的任何已优化函数的调用。

示例

当代码调用嵌入式操作系统所提供的函数时，如果该操作系统取代了 RVCT C 库所提供的函数，则将用 --library_interface=aeabi_clib 编译代码，以禁止调用由该操作系统替换的库函数的任何特殊 RVCT 变体。

另请参阅

- *《库和浮点支持指南》*中第1-3 页的*ARM 体系结构 ABI 遵从性*。

2.1.97 --library_type=*lib*

此选项启用要在链接时使用的所选库。

注意

将此选项与链接器一起使用可覆盖所有其他 --library_type 选项。

语法

--library_type=*lib*

其中，*lib* 是下列值之一：

standardlib　指定在链接时选择完整的 RVCT 运行时库。
使用此选项可在链接时利用全套的编译器和库优化。

microlib　指定在链接时选择 C 微型库 (microlib)。

缺省设置

如果未指定 --library_type，编译器将采用 --library_type=standardlib。

另请参阅

- *《库和浮点支持指南》*中第3-4 页的*使用 microlib 构建应用程序*。
- *《链接器参考指南》*中第2-34 页的*--library_type=lib*。

2.1.98 --licretry

如果使用的是浮动许可证，此选项会在调用 armcc 时最多尝试 10 次来获得许可证。

用法

典型的生成过程 （例如通宵生成）可能包含数以千计的 ARM 编译工具调用。每次工具调用都需要在客户端 （生成）计算机与许可证服务器之间进行网络通信。但是，如果当编译计算机尝试从许可证服务器获得许可证时发生临时性网络故障，工具可能无法获得许可证。因此，可以使用 --licretry 来尝试解决此类问题。

建议将此选项放入 RVCT40_CCOPT 系统变量中。这样便无需修改生成文件。

注意

只有在解决了与网络或许可证服务器设置有关的所有其他问题后，才能使用此选项。

另请参阅

- *《链接器参考指南》*中第2-35 页的*--licretry*
- *《汇编器指南》*中第3-2 页的*命令语法*
- *《实用程序指南》*中第2-32 页的*--licretry*
- *《RealView 编译工具要点指南》*中*第1-6 页的RVCT 使用的环境变量*
- 《ARM 工具 FLEXnet 许可证管理指南》

2.1.99 --list

此选项指示编译器为源文件生成原始列表信息。缺省情况下，原始列表文件的名称为具有文件扩展名 .lst 的输入文件的名称。

如果在命令行中指定了多个源文件，则仅为指定的前几个文件生成原始列表信息。

用法

原始列表信息通常用于生成带格式的列表。原始列表文件包含原始源代码行、包含文件转入转出信息以及由编译器生成的诊断消息。列表文件的每一行都由以下标识行类型的关键字母开头：

N　　常规源代码行。该行的其余部分是源代码行的文本。

X　　常规源代码行的扩展格式。该行的余下部分是文本。该行出现在 N 行后面，并且仅当该行包含非细微修改时才出现。注释被视为细微修改，而宏扩展、行接合以及三元组则被视为非细微修改。在扩展格式行中使用单一空格代替注释。

S　　通过 #if 或类似语句跳过的源代码行。该行的余下部分是文本。

注意

用 N 标记出结束跳跃的 #else、#elseif 或 #endif。

L　　指示源位置中的变更。即该行的格式类似于标识预处理程序的指令输出的 # 行：

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

其中 *key* 可以为：

1　　用于进入包含文件。

2　　用于从包含文件退出。

否则，将省略 *key*。原始列表文件中的第一行始终是标识主输入文件的 L 行。L 行也是 #line 指令的输出，其中省略了 *key*。L 行指示以下源代码行在原始列表文件中的源位置。

R/W/E　　表示诊断消息，其中：

R　　表示备注。

W　　表示警告。

E　　表示错误。

该行的格式为：

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

其中 *type* 可以为 R、W 或 E。

文件末尾处的错误指示主源文件的最后一行和 0 列号。

命令行错误是文件名为 "<command line>" 的错误。行号或列号不作为错误消息的一部分显示。

内部错误照常是带有位置信息的错误，并且消息文本以 (Internal fault) 开头。

当诊断消息显示一个列表 （例如，重载调用不明确时的所有竞争例程）时，初始诊断行将后跟具有相同完整格式的一行或多行。但代码字母是初始行中代码字母的小写版本。这些行中的源位置与相应的初始行相同。

示例

```
/* main.c */
#include <stdbool.h>
int main(void)
{
    return(true);
}
```

用 --list 选项编译此代码会生成原始列表文件：

```
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}
```

另请参阅

- 第2-15 页的*--asm*
- 第2-21 页的*-c*
- 第2-39 页的*--depend=filename*
- 第2-40 页的*--depend_format=string*
- 第2-72 页的*--info=totals*

- 第2-75 页的*--interleave*
- 第2-87 页的*--md*
- 第2-110 页的*-S*
- *《编译器用户指南》*中第6-3 页的*诊断消息的严重性*。

2.1.100 --list_macros

此选项用于在处理指定源文件后通过 stdout 列出宏定义。列出的输出中包含命令行上使用、编译器预定义以及在头文件和源文件中找到的宏定义，具体取决于用法。

用法

要列出命令行上已定义、编译器预定义以及在头文件和源文件中找到的宏，请将 --list_macros 与非空源文件一起使用。

要仅列出由编译器预定义和在命令行上指定的宏，请将 --list_macros 与空的源文件一起使用。

限制

禁止代码生成。

另请参阅

- 第4-115 页的*编译器预定义*
- 第2-35 页的*-Dname[(parm-list)][=def]*
- 第2-51 页的*-E*
- 第2-112 页的*--show_cmdline*
- 第2-128 页的*--via=filename*

2.1.101 --littleend

此选项指示编译器使用小端内存为 ARM 处理器生成代码。

在小端内存中，字的最低有效字节具有最低地址。

缺省设置

除非显式指定 --bigend，否则编译器将采用 --littleend。

另请参阅

- 第2-17 页的*--bigend*

2.1.102 --locale=*lang_country*

此选项可将源文件的缺省语言环境切换为在 *lang_country* 中指定的语言环境。

语法

--locale=*lang_country*

其中：

lang_country 是新的缺省语言环境。

请将此选项与 --multibyte_chars 结合使用。

限制

语言环境名称可能区分大小写，具体因主机平台而异。

由主机平台确定允许的语言环境设置。

确保安装了对主机平台的适当语言环境支持。

示例

例如，若要在基于英语的 Windows 工作站上编译日语源文件，请使用：

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

若要在基于英语的 UNIX 工作站上编译日语源文件，请使用：

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

另请参阅

- 第2-87 页的*--message_locale=lang_country[.codepage]*
- 第2-89 页的*--multibyte_chars*, *--no_multibyte_chars*

2.1.103 --loose_implicit_cast

此选项使非法隐式类型转换成为合法，如非零整数到指针的隐式类型转换。

示例

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

如果在编译此示例时未使用 --loose_implicit_cast 选项，则会生成错误。

如果用 --loose_implicit_cast 选项编译此示例，则会生成一条可禁止显示的警告消息。

2.1.104 --lower_ropi, --no_lower_ropi

此选项在使用 --apcs=/ropi 进行编译时启用或禁用限制较少的 C 代码。

缺省设置

缺省为 --no_lower_ropi。

注意

如果用 --lower_ropi 进行编译，则在 C 和 C++ 代码中，这种静态初始化在运行时由 C++ 构造函数机制完成。这使得这些静态初始化可使用 ROPI 代码。

另请参阅

- 第2-4 页的*--apcs=qualifer...qualifier*
- *--lower_rwpi, --no_lower_rwpi*
- *《编译器用户指南》*中第2-24 页的*位置无关限定符*。

2.1.105 --lower_rwpi, --no_lower_rwpi

此选项在使用 --apcs=/rwpi 进行编译时启用或禁用限制较少的 C 和 C++ 代码。

缺省设置

缺省为 --lower_rwpi。

注意

如果用 --lower_rwpi 进行编译，则这种静态初始化在运行时由 C++ 构造函数机制完成，即使在 C 代码中也是如此。这使得这些静态初始化可以使用 RWPI 代码。

另请参阅

- 第2-4 页的*--apcs=qualifer...qualifier*
- 第2-85 页的*--lower_ropi*, *--no_lower_ropi*
- *《编译器用户指南》*中第2-24 页的*位置无关限定符*。

2.1.106 --ltcg

此选项指示编译器以中间格式创建对象，以便执行链接时代码生成优化。应用的优化包括用于提高性能的跨模块内联，以及用于减少代码大小的基址共享。

注意

此选项可能会显著增加链接时间和内存请求。对于大型应用程序，建议在对象子集的部分链接步骤中生成代码。

示例

下面的示例演示如何使用 --ltcg 选项。

```
armcc -c --ltcg file1.c
armcc -c --ltcg file2.c
armlink --ltcg file1.o file2.o -o prog.axf
```

另请参阅

- 第2-90 页的*--multifile*, *--no_multifile*
- 第2-94 页的*-Onum*
- *《链接器参考指南》*中第2-36 页的*--ltcg*

2.1.107 -M

此选项指示编译器生成适合 make 实用程序使用的 makefile 相关性行列表。

编译器仅执行编译的预处理程序步骤。缺省情况下，输出位于标准输出流上。

如果指定多个源文件，则将创建单个相关性文件。

如果指定 -o *filename* 选项，则在标准输出上生成的相关性行引用的是 filename.o 而不是 source.o。不过，使用 -M -o *filename* 组合时不会生成对象文件。

使用 --md 选项可以为每个源文件生成相关性行和对象文件。

示例

可使用标准 UNIX 和 MS-DOS 记号将输出重定向到文件，例如：

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

另请参阅

- 第2-21 页的*-C*
- 第2-39 页的*--depend=filename*
- 第2-41 页的*--depend_system_headers*, *--no_depend_system_headers*
- 第2-51 页的*-E*
- *--md*
- 第2-92 页的*-o filename*

2.1.108 --md

此选项指示编译器编译源文件并将 makefile 相关性行写入文件中。

输出文件适合由 make 实用程序使用。

编译器将源文件命名为 *filename*.d，其中 *filename* 是源文件的名称。如果指定多个源文件，则会为每个源文件创建一个相关性文件。

另请参阅

- 第2-39 页的*--depend=filename*
- 第2-40 页的*--depend_format=string*
- 第2-41 页的*--depend_system_headers*, *--no_depend_system_headers*
- 第2-86 页的*-M*
- 第2-92 页的*-o filename*

2.1.109 --message_locale=*lang_country*[.*codepage*]

此选项可将错误消息和警告消息的缺省显示语言切换为在 *lang_country* 或 *lang_country.codepage* 中所指定的语言。

语法

--message_locale=*lang_country*[.*codepage*]

其中：

lang_country[.*codepage*]

是错误消息和警告消息的缺省显示语言。

允许的语言与主机平台无关。

支持下列设置：

- en_US
- zh_CN
- ko_KR
- ja_JP。

缺省设置

如果未指定 --message_locale，编译器将采用 --message_locale=en_US。

限制

确保安装了对主机平台的适当语言环境支持。

语言环境名称可能区分大小写，具体因主机平台而异。

根据主机平台的不同，能够指定代码页及其含义。

错误

如果指定了不受支持的设置，编译器会生成一条错误消息。

示例

若要用日语显示消息，请使用：

```
--message_locale=ja_JP
```

另请参阅

- 第2-84 页的*--locale=lang_country*
- 第2-89 页的*--multibyte_chars*, *--no_multibyte_chars*

2.1.110 --min_array_alignment=*opt*

此选项使您可以指定数组的最低对齐要求。

语法

--min_array_alignment=*opt*

其中：

opt　　指定数组的最低对齐要求。*opt* 的值为：

1　　字节对齐或未对齐

2　　双字节 （半字）对齐

4　　四字节 （字）对齐

8　　八字节 （双字）对齐。

缺省设置

如果未指定 --min_array_alignment 选项，编译器将采用 --min_array_alignment=1。

示例

使用 --min_array_alignment=8 编译以下代码时，将提供注释中所述的对齐方式：

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

另请参阅

- 第4-2 页的*__align*
- 第4-4 页的*__ALIGNOF__*

2.1.111 --mm

此选项与 -M --no_depend_system_headers 的作用相同。

另请参阅

- 第2-41 页的*--depend_system_headers, --no_depend_system_headers*
- 第2-86 页的*-M*

2.1.112 --multibyte_chars, --no_multibyte_chars

此选项启用或禁用注释、字符串以及字符常数中多字节字符序列的处理。

缺省设置

缺省为 --no_multibyte_chars。

用法

多字节编码适用于日语 *Shift-Japanese Industrial Standard* (Shift-JIS) 之类的字符集。

另请参阅

- 第2-84 页的*--locale=lang_country*
- 第2-87 页的*--message_locale=lang_country[.codepage]*

2.1.113 --multifile, --no_multifile

此选项启用或禁用多文件编译。

如果选择 --multifile，则编译器将对命令行中指定的所有文件 （而不是单个文件）执行优化。会将指定的文件编译成一个对象文件。

在命令行中指定的第一个源文件之后命名组合对象文件。若要为组合对象文件指定其他名称，请使用 -o *filename* 选项。

为满足标准 make 系统的要求，将为命令行中指定的每个后续源文件创建一个空对象文件。

——注意——

如果在命令行中指定了单个源文件，则用 --multifile 进行编译将无效。

缺省设置

如果未指定 -O3，则缺省为 --no_multifile。

如果指定了 -O3 选项，则缺省为 --multifile。

用法

如果选择 --multifile，则编译器可通过编译多个源文件来执行额外优化。

虽然对可在命令行中指定的源文件数没有限制，但一般不要超过 10 个文件，因为 --multifile 在编译时需要占用大量内存。若要获得最佳优化效果，请选择在功能上相关的一小组源文件。

示例

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

所生成的对象文件将被命名为 test.o 而不是 test1.c，并将为命令行中指定的每个源文件 test1.c ... testn.c 创建对应的空对象文件 test2.o 到 testn.o。

另请参阅

- 第2-21 页的*-c*
- 第2-37 页的*--default_extension=ext*
- 第2-86 页的*--ltcg*
- 第2-92 页的*-o filename*
- 第2-94 页的*-Onum*
- 第2-132 页的*--whole_program*

2.1.114 --multiply_latency=*cycles*

此选项将向编译器提示硬件乘法器所使用的周期数。

语法

--multiply_latency=*cycles*

其中，*cycles* 是使用的周期数。

用法

使用此选项可向编译器通知 MUL 指令使用乘法器块和芯片的相关部件的周期数。在该指令完成之前，芯片的这些部件将无法用于其他指令，要使用的任何后续指令也无法使用 MUL 的结果。

处理器可能具有两个或更多个为给定硬件实现而设置的乘法器选项。例如，可能将一个实现配置为需要一个周期来执行。另一实现可能需要 33 个周期来执行。此选项用于为给定处理器传达正确的周期数。

示例

```
--multiply_latency=33
```

另请参阅

- *《Cortex™-M1 技术参考手册》*。

2.1.115 --nonstd_qualifier_deduction, --no_nonstd_qualifier_deduction

此选项控制是否要在 C++ 中限定名的限定符部分内执行非标准模板自变量推算。

如果启用此功能，则可在类似 `A<T>::B` 或 `T::B` 的上下文中推算模板参数 `T` 的模板自变量。标准推算机制将它们视为以下非推算上下文：所用值属于显式指定模板参数或在别处推算的模板参数。

注意

`--nonstd_qualifier_deduction` 仅作为不符合 C++ 标准的旧式源代码的迁移辅助选项。不建议使用此选项。

模式

仅当源语言为 C++ 时，此选项才有效。

缺省设置

缺省为 `--no_nonstd_qualifier_deduction`。

2.1.116 -o *filename*

此选项指定输出文件的名称。所生成的输出文件的完整名称因所使用的组合选项而异，具体说明详见第2-92 页的表 2-5 和第2-93 页的表 2-6。

语法

如果指定了 -o 选项，则编译器将根据表 2-5 的约定命名输出文件。

表 2-5 用 -o 选项进行编译

编译器选项	操作	使用说明
-o-	将输出写入标准输出流	*filename* 是 -。除非指定了 -E，否则将采用 -S。
-o *filename*	生成名为 *filename* 的可执行映像	
-c -o *filename*	生成名为 *filename* 的对象文件	
-S -o *filename*	生成名为 *filename* 的汇编语言文件	
-E -o *filename*	生成包含预处理程序输出的名为 *filename* 的文件	

如果未指定 -o 选项，则编译器将根据表 2-6 的约定命名输出文件。

表 2-6 编译时未使用 -o 选项

编译器选项	操作	使用说明
-c	生成一个对象名称，其缺省名称是具有文件扩展名 .o 的输入文件名称。	
-S	生成一个输出文件，其缺省名称是具有文件扩展名 .s 的输入文件名称。	
-E	将预处理程序的输出写入标准输出流	
（无选项）	生成缺省名称为 __image.axf 的可执行映像	-o、-c、-E 和 -S 都不在命令行中指定

注意

此选项将覆盖 --default_extension 选项。

另请参阅

- 第2-15 页的*--asm*
- 第2-21 页的*-c*
- 第2-37 页的*--default_extension=ext*
- 第2-39 页的*--depend=filename*
- 第2-40 页的*--depend_format=string*
- 第2-51 页的*-E*
- 第2-75 页的*--interleave*
- 第2-80 页的*--list*
- 第2-87 页的*--md*
- 第2-110 页的*-S*

2.1.117 -O*num*

此选项指定要在编译源文件时使用的优化级别。

语法

-O*num*

其中 *num* 是下列值之一：

0
: 最低优化。关闭大多数优化。它提供可能的最佳调试视图和最低优化级别。

1
: 受限优化。删除未使用的内联函数和未使用的静态函数关闭严重影响调试视图的优化。如果与 --debug 一起使用，则此选项将提供具有良好代码密度且令人满意的调试视图。

2
: 高度优化。如果与 --debug 一起使用，则调试视图可能不会令人满意，因为对象代码到源代码的映射有时会不清晰。

  这是缺省设置。

3
: 最大优化。-O3 执行与 -O2 相同的优化，但与 -O2 相比，前者生成的代码中空间与时间优化之间的平衡更侧重于空间或时间。也就是说：

  - -O3 -Otime 旨在比 -O2 -Otime 更快速地生成代码，但这是以增加映像大小作为代价的
  - -O3 -Ospace 旨在比 -O2 -Ospace 生成更少的代码，但可能会降低性能。

此外，-O3 还会执行更积极的额外优化，如：

- 高级标量优化，其中包括 -O3 -Otime 的循环展开。这以很小的代码大小开销就会获得显著的性能，但编译时间会较长。
- -O3 -Otime 的更积极的内联和自动内联。
- 缺省情况下的多文件编译。

注意

如果使用 --fpmode 选项选择适当的数值模型，则可能会影响浮点代码的性能。

注意

不要依赖这些优化的实现细节，因为在将来的版本中它们可能会发生变化。

缺省设置

如果未指定 -O*num*，则编译器将采用 -O2。

另请参阅

- 第2-16 页的*--autoinline, --no_autoinline*
- 第2-36 页的*--debug, --no_debug*
- 第2-58 页的*--forceinline*
- 第2-59 页的*--fpmode=model*
- 第2-73 页的*--inline, --no_inline*
- 第2-86 页的*--ltcg*
- 第2-90 页的*--multifile, --no_multifile*
- 第2-96 页的*-Ospace*
- 第2-96 页的*-Otime*
- *《编译器用户指南》*中第5-2 页的*优化代码*

2.1.118 --old_specializations, --no_old_specializations

此选项控制是否接受 C++ 中的旧式模板特化。

旧式模板特化不使用 template<> 语法。

注意

--old_specializations 选项仅作为不符合 C++ 标准的旧式源代码的迁移辅助选项。不建议使用此选项。

模式

仅当源语言为 C++ 时，此选项才有效。

缺省设置

缺省为 --no_old_specializations。

2.1.119 -Ospace

此选项指示编译器执行减小映像大小的优化，这是以可能延长执行时间为代价的。

如果代码大小比性能更重要，请使用此选项。例如，如果选择 -Ospace 选项，则将由外联函数调用而非内联代码执行大型结构副本。

如果需要，则可以使用 -Otime 编译对时间要求严格的代码部分，而使用 -Ospace 编译其余部分。

缺省设置

如果未指定 -Ospace 或 -Otime，编译器将采用 -Ospace。

另请参阅

- 第2-96 页的*-Otime*
- 第2-94 页的*-Onum*
- 第4-64 页的*#pragma Onum*
- 第4-65 页的*#pragma Ospace*
- 第4-65 页的*#pragma Otime*

2.1.120 -Otime

此选项指示编译器执行减少执行时间的优化，这是以可能增加映像大小为代价的。

如果执行时间比代码大小更重要，请使用此选项。如果需要，则可以使用 -Otime 编译对时间要求严格的代码部分，而使用 -Ospace 编译其余部分。

缺省设置

如果未指定 -Otime，编译器将采用 -Ospace。

示例

如果选择 -Otime 选项，则编译器会将以下代码：

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

编译为：

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

另请参阅

- 第2-90 页的*--multifile, --no_multifile*
- 第2-94 页的*-Onum*
- 第2-96 页的*-Ospace*
- 第4-64 页的*#pragma Onum*
- 第4-65 页的*#pragma Ospace*
- 第4-65 页的*#pragma Otime*

2.1.121 --parse_templates, --no_parse_templates

此选项启用或禁用 C++ 中通用格式 （即在定义模板时和对其进行实例化之前）的非类模板的分析。

注意

. --no_parse_templates 选项仅作为不符合 C++ 标准的旧式源代码的迁移辅助选项。不建议使用此选项。

模式

仅当源语言为 C++ 时，此选项才有效。

缺省设置

缺省为 --parse_templates。

注意

--no_parse_templates 不能与 --dep_name 一起使用，因为如果启用从属名称处理，则缺省情况下将执行分析。如果一起使用这两个选项，则会生成错误。

另请参阅

- 第2-38 页的*--dep_name, --no_dep_name*
- 第5-14 页的*模板实例化*

2.1.122 --pch

此选项指示编译器在 PCH 文件存在的情况下使用该文件，在该文件不存在的情况下创建一个这样的文件。

如果指定 --pch 选项，则编译器将搜索名为 *filename*.pch 的 PCH 文件，其中 *filename*.* 是主源文件的名称。如果 PCH 文件 *filename*.pch 存在，则编译器将使用该文件；否则，将在主源文件所在的同一目录中创建名为 *filename*.pch 的 PCH 文件。

限制

如果在同一命令行中包含 --use_pch=*filename* 选项或 --create_pch=*filename* 选项，则此选项将无效。

另请参阅

- 第2-34 页的*--create_pch=filename*
- *--pch_dir=dir*
- 第2-99 页的*--pch_messages*, *--no_pch_messages*
- 第2-99 页的*--pch_verbose*, *--no_pch_verbose*
- 第2-125 页的*--use_pch=filename*
- 第4-62 页的*#pragma hdrstop*
- 第4-63 页的*#pragma no_pch*
- *《编译器用户指南》*中第2-18 页的*预编译的头文件*。

2.1.123 --pch_dir=*dir*

此选项用于指定 PCH 文件的存储目录。每次创建或访问 PCH 文件时，都可以访问此目录。

可在自动或手动 PCH 模式下使用此选项。

语法

--pch_dir=*dir*

其中：

dir 是 PCH 文件的存储目录的名称。

错误

如果指定的目录 *dir* 不存在，则编译器会生成错误。

另请参阅

- 第2-34 页的*--create_pch=filename*
- 第2-98 页的*--pch*
- 第2-99 页的*--pch_messages*, *--no_pch_messages*
- 第2-99 页的*--pch_verbose*, *--no_pch_verbose*
- 第2-125 页的*--use_pch=filename*
- 第4-62 页的*#pragma hdrstop*
- 第4-63 页的*#pragma no_pch*
- *《编译器用户指南》*中第2-18 页的*预编译的头文件*。

2.1.124 --pch_messages, --no_pch_messages

此选项允许或禁止显示特定消息，此类消息指出在当前编译中使用了 PCH 文件。

缺省设置

缺省为 --pch_messages。

另请参阅

- 第2-34 页的*--create_pch=filename*
- 第2-98 页的*--pch*
- 第2-98 页的*--pch_dir=dir*
- *--pch_verbose*, *--no_pch_verbose*
- 第2-125 页的*--use_pch=filename*
- 第4-62 页的*#pragma hdrstop*
- 第4-63 页的*#pragma no_pch*
- *《编译器用户指南》*中第2-18 页的*预编译的头文件*。

2.1.125 --pch_verbose, --no_pch_verbose

此选项允许或禁止显示特定消息，此类消息指明不能预编译文件的原因。

在自动 PCH 模式中，此选项可确保对于每个不能用于当前编译的 PCH 文件，都显示一条消息来指明不能使用该文件的原因。

缺省设置

缺省为 --no_pch_verbose。

另请参阅

- 第2-34 页的*--create_pch=filename*
- 第2-98 页的*--pch*
- 第2-98 页的*--pch_dir=dir*
- *--pch_messages*, *--no_pch_messages*
- 第2-125 页的*--use_pch=filename*
- 第4-62 页的*#pragma hdrstop*
- 第4-63 页的*#pragma no_pch*
- *《编译器用户指南》*中第2-18 页的*预编译的头文件*。

2.1.126 --pending_instantiations=*n*

此选项指定 C++ 中模板的最大并发实例化数。

语法

`--pending_instantiations=`*n*

其中：

n 是允许的最大并发实例化数。
如果 *n* 为零，则没有限制。

模式

仅当源语言为 C++ 时，此选项才有效。

缺省设置

如果未指定 `--pending_instantations` 选项，编译器将采用 `--pending_instantiations=64`。

用法

使用此选项可检测失控递归实例化。

2.1.127 --phony_targets

此选项指示编译器生成虚拟 makefile 规则。如果删除头文件而未对 makefile 进行相应更新，则这些规则可清除 `make` 错误。

此选项与 GCC 命令行选项 `-MP` 类似。

示例

示例输出：

```
source.o: source.c
source.o: header.h
header.h:
```

另请参阅

- 第2-39 页的*--depend=filename*
- 第2-40 页的*--depend_format=string*
- 第2-41 页的*--depend_system_headers*, *--no_depend_system_headers*
- 第2-42 页的*--depend_target=target*
- 第2-70 页的*--ignore_missing_headers*
- 第2-86 页的*-M*
- 第2-87 页的*--md*

2.1.128 --pointer_alignment=*num*

此选项指定应用程序所需的未对齐指针支持。

语法

--pointer_alignment=*num*

其中 *num* 是下列值之一：

1 将经由指针的访问视为具有对齐方式 1，即字节对齐或未对齐。

2 将经由指针的访问视为具有不超过 2 的对齐方式，即最多半字对齐。

4 将经由指针的访问视为有不超过 4 的对齐方式，即最多字对齐。

8 经由指针的访问具有常规对齐，即最多双字对齐。

用法

此选项可在不需要对齐的情况下，帮助移植为体系结构编写的源代码。使用 __packed 限定符可更精细地控制对未对齐数据的访问，并且对所生成代码的质量影响较小。

限制

即使在支持未对齐访问的 CPU 上，不对齐指针也可能增加代码大小。这是因为只有加载和存储指令的子集才会从未对齐访问支持中受益。编译器不能直接对未对齐内存对象使用多字传送或协处理器内存传送，包括硬件浮点加载和存储。

注意

- 当对不具备未对齐访问的硬件支持的 CPU （如 v6 以前版本的体系结构）进行编译时，代码大小可能会显著增加。
- 此选项既不影响对象在内存中的放置，也不影响结构的布局与填充。

另请参阅

- 第4-11 页的*__packed*
- 第4-65 页的*#pragma pack(n)*
- *《编译器用户指南》*中第5-23 页的*对齐数据*。

2.1.129 --preinclude=*filename*

此选项指示编译器在编译开头包括指定文件的源代码。

语法

`--preinclude=`*filename*

其中：

filename　　是要包含其源代码的文件的名称。

用法

此选项可用于建立标准宏定义。在包含搜索列表上的目录中搜索 *filename*。

可以在命令行上重复指定此选项。此结果按照指定的顺序预先包括在文件中。

示例

```
armcc --preinclude file1.h --preinclude file2.h -c source.c
```

另请参阅

- 第2-69 页的*-Idir[,dir,...]*
- 第2-76 页的*-Jdir[,dir,...]*
- 第2-76 页的*--kandr_include*
- 第2-117 页的*--sys_include*
- *《编译器用户指南》*中第2-15 页的*头文件*

2.1.130 --preprocessed

此选项强制预处理程序像处理已被替换的宏那样处理文件扩展名为 .i 的文件。

用法

此选项为您提供了使用不同预处理程序的机会。生成经过预处理的代码，然后将这些代码以 *filename*.i 的形式提供给编译器，同时使用 --preprocessed 通知编译器该文件已被预处理。

限制

此选项仅适用于宏。三元组、行连接、注释和所有其他预处理程序项目都由预处理程序按正常方式进行预处理。

如果您使用 --compile_all_input， .i 文件将被视为 .c 文件。预处理程序就当事先未进行任何预处理一样工作。

示例

```
armcc --preprocessed foo.i -c -o foo.o
```

另请参阅

- 第2-23 页的*--compile_all_input*, *--no_compile_all_input*
- 第2-51 页的*-E*

2.1.131 --profile=*filename*

此选项指示编译器使用 ARM Profiler 的反馈，以生成更小且性能更高的代码。

语法

--profile=*filename*

其中：

filename　　是 ARM Profiler 分析文件的名称。

示例

本示例使用生成 hello.c 的代码时在 hello_001.apa 中提供的 ARM Profiler 反馈。

```
armcc -c -O3 -Otime --profile=hello_001.apa hello.c
```

另请参阅

- *《ARM Profiler 用户指南》*。

2.1.132 --project=*filename*, --no_project=*filename*

--project=*filename* 选项指示编译器加载 *filename* 所指定的项目模板文件。

注意

若要使用 *filename* 作为缺省项目文件，请将 RVDS_PROJECT 环境变量设置为 *filename*。

--no_project 选项禁止使用环境变量 RVDS_PROJECT 所指定的缺省项目模板文件。

语法

--project=*filename*

--no_project

其中：

filename 是项目模板文件的名称。

限制

仅当项目模板文件中的选项与命令行中已设置的选项不发生冲突时，才会设置前者。如果项目模板文件中的选项与现有命令行选项发生冲突，则后者优先。

示例

请考虑以下项目模板文件：

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

当将 RVDS_PROJECT 环境变量设置为指向此文件时，以下命令：

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

将生成以下实际命令行：

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

另请参阅

- 第2-107 页的*--reinitialize_workdir*
- 第2-132 页的*--workdir=directory*

2.1.133 --reassociate-saturation

此选项通过允许重新关联饱和算法，在向量化使用饱和相加的循环时启用更积极的优化。

限制

饱和相加不具关联性，因此启用重新关联将会导致准确度降低。

示例

除非指定了 --reassociate-saturation，否则，下列代码将不执行向量化。

```
#include <dspfns.h>
int f(short *a, short *b)
{
    int i;
    int r = 0;
    for (i = 0; i < 100; i++)
         r=L_mac(r,a[i],b[i]);
    return r;
}
```

2.1.134 --reduce_paths, --no_reduce_paths

此选项允许或禁止删除文件路径中的冗余路径名信息。

如果允许删除冗余路径名信息，则编译器会从传递给操作系统的目录中删除 xyz\.. 格式的序列。这包括编译器在执行 #include 搜索等操作时自身构建的系统路径。

注意

如果 xyz 为链接，则 xyz\.. 格式序列的删除可能无效。

模式

此选项仅在 Windows 系统中有效。

用法

Windows 系统对文件路径有 260 个字符的限制。如果存在其绝对路径名长度超过 260 个字符的路径名，则可以使用 --reduce_paths 选项，该选项通过将目录与对应的 .. 实例相匹配并成对删除 directory/.. 序列，可以缩短绝对路径名的长度。

注意

建议优先使用 --reduce_paths 选项来尽量缩短路径长度，并避免使用长文件路径和深层嵌套文件路径。

缺省设置

缺省为 --no_reduce_paths。

示例

编译以下文件

```
..\..\..\xyzzy\xyzzy\objects\file.c
```

该文件位于以下目录中

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

这会生成以下实际的路径

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

如果使用 option --reduce_paths 选项从同一目录中编译同一文件，则会生成以下一个实际的路径

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

2.1.135 --reinitialize_workdir

此选项使您可以使用 --workdir 重新初始化项目模板工作目录。

当使用 --workdir 设置的目录引用包含已修改项目模板文件的现有工作目录时，指定此选项时会导致删除该工作目录并用原始项目模板文件的新副本重新创建该目录。

限制

此选项必须与 --workdir 选项结合使用。

另请参阅

- 第2-104 页的*--project=filename, --no_project=filename*
- 第2-132 页的*--workdir=directory*

2.1.136 --relaxed_ref_def, --no_relaxed_ref_def

此选项允许多个对象文件使用全局变量的试验定义。某些传统程序是用此声明样式编写的。

用法

此选项主要是为了实现与 GNU C，不建议用于新应用程序代码。

缺省设置

缺省为严格引用和定义。（每个全局变量只能在一个对象文件中声明。）但是，如果在命令行中指定 ARM Linux 配置文件并使用 --translate_gcc，则缺省为 --relaxed_ref_def。

限制

此选项在 C++ 中无效。

另请参阅

- 第2-10 页的*--arm_linux_config_file=path*
- 第2-11 页的*--arm_linux_configure*
- 第2-120 页的*--translate_gcc*
- 《国际标准原理 - 编程语言 - C》 (*Rationale for International Standard - Programming Languages - C*)。

2.1.137 --remarks

此选项指示编译器发出备注消息，如结构中的填充警告。

注意

缺省情况下，编译器不发出备注。

另请参阅

- 第2-19 页的*--brief_diagnostics, --no_brief_diagnostics*
- 第2-44 页的*--diag_error=tag[,tag,...]*
- 第2-45 页的*--diag_remark=tag[,tag,...]*
- 第2-45 页的*--diag_style={arm|ide|gnu}*
- 第2-46 页的*--diag_suppress=tag[,tag,...]*
- 第2-48 页的*--diag_warning=tag[,tag,...]*
- 第2-52 页的*--errors=filename*
- 第2-129 页的*-W*
- 第2-133 页的*--wrap_diagnostics, --no_wrap_diagnostics*

2.1.138 --restrict, --no_restrict

此选项允许或禁止使用 C99 关键字 restrict。

注意

支持将备选关键字 __restrict 和 __restrict__ 作为 **restrict** 的同义词。无论是否使用 --restrict 选项，这些备选关键字都始终可用。

缺省设置

当编译 ISO C99 源代码时，缺省选项将允许使用 C99 关键字 **restrict**。

当编译 ISO C90 或 ISO C++ 源代码时，缺省情况下，将禁止使用 C99 关键字 **restrict**。

另请参阅

- 第3-7 页的*restrict*

2.1.139 --retain=*option*

使用此选项可以限制编译器执行的优化，并且在执行验证、调试和覆盖率测试时可能会有用。

语法

--retain=*option*

其中， *option* 是下列值之一：

fns 防止删除未使用的函数

inlinefns 防止删除未使用的内联函数

noninlinefns 防止删除未使用的非内联函数

paths 防止进行路径删除优化，如 a||b 转换为 a|b。它支持*修正条件判定覆盖* (MCDC) 测试。

calls 防止通过内联或尾调用等删除调用。

calls:distinct
防止通过交叉跳转等合并调用 （即通用尾路径合并）。

libcalls 防止通过内联扩展等删除对库函数的调用。

数据 防止删除数据。

rodata 防止删除只读数据。

rwdata 防止删除读写数据。

data:order 防止对数据重新排序。

另请参阅

- 第4-35 页的*__attribute__((nomerge))*
- 第4-37 页的*__attribute__((notailcall))*

2.1.140 --rtti, --no_rtti

此选项控制对 C++ 中 RTTI 功能 dynamic_cast 和 typeid 的支持。

模式

仅当源语言为 C++ 时，此选项才有效。

缺省设置

缺省选项为 --rtti。

另请参阅

- 第2-50 页的*--dllimport_runtime*, *--no_dllimport_runtime*

2.1.141 -S

此选项指示编译器将其生成的机器代码的反汇编输出到文件中。

与 --asm 选项不同的是，此选项不生成对象模块。汇编输出文件的缺省名称为当前目录中的 *filename*.s，其中 *filename* 是去掉任何前导目录名的源文件名称。用 -o 选项可覆盖该缺省文件名。

可以使用 armasm 汇编输出文件并生成对象代码。编译器会为 AAPCS 变体以及字节顺序等命令行选项添加 ASSERT 指令，以确保在重新汇编输出时使用兼容的编译器选项和汇编器选项。为汇编器和编译器指定的 AAPCS 设置必须相同。

另请参阅

- 第2-4 页的*--apcs=qualifer...qualifier*
- 第2-15 页的*--asm*
- 第2-21 页的*-c*
- 第2-72 页的*--info=totals*
- 第2-75 页的*--interleave*
- 第2-80 页的*--list*
- 第2-92 页的*-o filename*
- *《汇编器指南》*。

2.1.142 --shared

此选项允许在使用 --arm_linux_paths 选项为 ARM Linux 进行编译时生成共享库。它允许根据 ARM Linux 配置，选择适于在共享库中使用的库和初始化代码。

限制

此选项必须与 --arm_linux_paths 和 --apcs=/fpic 结合使用。

示例

将两个对象文件 obj1.o 和 obj2.o 链接到共享库 libexample.o 中：

```
armcc --arm_linux_paths --arm_linux_config_file=my_config_file --shared -o
libexample.so obj1.o obj2.o
```

另请参阅

- 第2-8 页的*--arm_linux*
- 第2-10 页的*--arm_linux_config_file=path*
- 第2-11 页的*--arm_linux_configure*
- 第2-13 页的*--arm_linux_paths*
- 第2-24 页的*--configure_cpp_headers=path*
- 第2-25 页的*--configure_extra_includes=paths*
- 第2-26 页的*--configure_extra_libraries=paths*
- 第2-27 页的*--configure_gcc=path*
- 第2-28 页的*--configure_gld=path*
- 第2-29 页的*--configure_sysroot=path*
- 第2-118 页的*--translate_g++*
- 第2-120 页的*--translate_gcc*
- 第2-121 页的*--translate_gld*
- *《链接器参考指南》*中第2-3 页的*--arm_linux*
- *《链接器参考指南》*中第2-33 页的*--library=name*
- *《链接器参考指南》*中第2-51 页的*--[no_]search_dynamic_libraries*
- 《应用程序注释 212 — 使用 RVCT 4.0 版以及 GNU 工具和库生成 Linux 应用程序》(*Application Note 212 - Building Linux applications using RVCT v4.0 and the GNU Tools and Libraries*)。

2.1.143 --show_cmdline

此选项显示处理命令行选项的方式。

命令以其首选形式显示，并展开所有 via 文件的内容。

另请参阅

- 第2-2 页的*-Aopt*
- 第2-77 页的*-Lopt*
- 第2-128 页的*--via=filename*

2.1.144 --signed_bitfields, --unsigned_bitfields

此选项使类型为 **int** 的位域带符号或无符号。

C 标准规定：如果在声明位域时所使用的类型说明符是 **int**，或者是定义为 **int** 的 **typedef** 名称，则位域是带符号还是无符号将取决于实现情况。

缺省设置

缺省为 --unsigned_bitfields。但是，如果在命令行中指定 ARM Linux 配置文件并使用 --translate_gcc 或 --tranlsate_g++，则缺省为 --signed_bitfields。

注意

在 AAPCS 标准的 2.03 版中，放松了对 ARM 上位域缺省情况下无符号的要求。

示例

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

用 --signed_bitfields 编译此代码会将位域视为有符号位域。

另请参阅

- 第2-10 页的*--arm_linux_config_file=path*
- 第2-11 页的*--arm_linux_configure*

2.1.145 --signed_chars, --unsigned_chars

此选项使 **char** 类型带符号或无符号。

如果 **char** 带符号，则编译器将定义宏 __FEATURE_SIGNED_CHAR。

注意

如果要混合分别使用和不使用此选项编译的转换单元，并且转换单元共用接口或数据结构，则必须格外小心。

ARM ABI 将 **char** 定义为无符号字节，这是 RVCT 附带提供的 C++ 库使用的解释。

缺省设置

缺省选项为 --unsigned_chars。

另请参阅

- 第4-115 页的*预定义宏*

2.1.146 --split_ldm

此选项指示编译器将 LDM 和 STM 指令分别拆分为两个或更多的 LDM 或 STM 指令。

如果选择 --split_ldm，则 LDM 或 STM 指令的最大寄存器传送数将根据下列情况限定为相应的值：

- 对于所有 STM，为 5
- 对于未加载 PC 的 LDM，为 5
- 对于加载 PC 的 LDM，为 4。

如果所需的寄存器传送数超过上述这些限制，则应使用多个 LDM 或 STM 指令。

用法

使用 --split_ldm 选项可在满足下列条件的 ARM 系统上缩短中断等待时间：

- 没有高速缓存或写缓冲器 （如没有高速缓存的 ARM7TDMI）
- 使用零等待状态，并具有 32 位内存。

注意

使用 --split_ldm 会增加代码大小，并且会使性能略微有所降低。

限制

- 使用 --split_ldm 时，缺省情况下将拆分内联汇编器的 LDM 和 STM 指令。但编译器以后可能会将这些单独的指令重新合并为一条 LDM 或 STM 指令。
- 使用 --split_ldm 时，仅拆分 LDM 和 STM 指令。
- 有些目标硬件无法从用 --split_ldm 编译的代码中获益。例如：
  - 对高速缓存系统或具有写缓冲器的处理器没有明显好处。

— 对具有非零等待状态内存的系统或具有慢速外围设备的系统没有好处。在此类系统中，中断等待时间是由速度最慢的内存或外设访问所需的周期数决定的。这一延迟时间一般远大于由多个寄存器传送产生的延迟时间。

另请参阅

- *《编译器用户指南》*中第7-9 页的*指令扩展*。

2.1.147 --split_sections

此选项指示编译器为源文件中的每个函数分别生成一个 ELF 节。

按生成输出节的函数的名称命名输出节，且带有 i. 前缀。

注意

如果要将特定的数据项或结构放在单独的节中，请用 `__attribute__((section(...)))` 分别对其进行标记。

如果要删除未使用的函数，建议对此选项优先使用链接器反馈优化。这是因为链接器反馈通过避免产生拆分所有节的开销，可生成较短的代码。

限制

此选项降低了在函数之间共享地址、数据和字符串的可能性。因此，可能会使某些函数的代码大小略有增加。

示例

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

用 --split_sections 编译此代码会生成以下结果：

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

另请参阅

- 第2-36 页的*--data_reorder, --no_data_reorder*
- 第2-56 页的*--feedback=filename*
- 第2-90 页的*--multifile, --no_multifile*
- 第4-38 页的*__attribute__((section("name")))*
- 第4-56 页的*#pragma arm section [section_sort_list]*
- *《编译器用户指南》*中第2-26 页的*使用链接器反馈*

2.1.148 --strict, --no_strict

此选项根据所使用的源语言选项，强制实施或放宽严格 C 或严格 C++。

如果选择 --strict，则会执行下列操作：

- 禁用与 ISO C 或 ISO C++ 发生冲突的功能
- 在使用非标准功能时返回错误消息。

缺省设置

缺省选项为 --no_strict。

用法

--strict 强制遵循下列标准：

ISO C90
- ISO/IEC 9899:1990， 1990 C 国际标准。
- ISO/IEC 9899 AM1， 1995 标准附录 1。

ISO C99 ISO/IEC 9899:1999， 1999 C 国际标准。

ISO C++ ISO/IEC 14822:2003， 2003 C++ 国际标准。

错误

当 --strict 生效时，如果违反了相关的 ISO 标准，则编译器会发出错误消息。

可以用常规方式控制诊断消息的严重性。

示例

```
void foo(void)
{
    long long i; /* okay in nonstrict C90 */
}
```

用 --strict 编译此代码时会生成错误。

另请参阅

- 第2-22 页的*--c90*
- 第2-22 页的*--c99*
- 第2-30 页的*--cpp*
- 第2-66 页的*--gnu*
- *--strict_warnings*
- 第3-12 页的*标识符中的美元符号*
- *《编译器用户指南》*中第1-4 页的*源语言模式*。

2.1.149 --strict_warnings

--strict 模式中是错误的诊断将在可能时降级为警告。有时，编译器无法降级某一严格错误，例如，无法构造要恢复的合法程序时。

错误

当 --strict_warnings 生效时，如果违反了相关 ISO 标准，则编译器通常会发出警告消息。

可以用常规方式控制诊断消息的严重性。

注意

在某些情况下，如果编译器检测到严重非法行为，则会发出错误消息而不是警告，并会终止编译。例如：

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

如果不使用 --dollar 选项，则使用 --strict_warnings 编译此代码时会生成错误。

示例

```
void foo(void)
{
    long long i; /* okay in nonstrict C90 */
}
```

用 --strict_warnings 编译此代码时会生成警告消息。

编译会继续执行，即使表达式 **long long** 严重非法也是如此。

另请参阅

- 第1-3 页的*源语言模式*
- 第3-12 页的*标识符中的美元符号*
- 第2-22 页的*--c90*
- 第2-22 页的*--c99*
- 第2-30 页的*--cpp*
- 第2-66 页的*--gnu*
- 第2-115 页的*--strict*, *--no_strict*

2.1.150 --sys_include

此选项从包含搜索路径中删除当前位置。

带引号的包含文件的处理方式与带尖括号的包含文件类似，但前者总是先在 -I 指定的目录中搜索，而后者先在 -J 目录中搜索。

另请参阅

- 第2-69 页的*-Idir[,dir,...]*
- 第2-76 页的*-Jdir[,dir,...]*
- 第2-76 页的*--kandr_include*
- 第2-102 页的*--preinclude=filename*
- *《编译器用户指南》*中第2-15 页的*当前位置*
- *《编译器用户指南》*中第2-16 页的*搜索路径*。

2.1.151 --thumb

此选项将编译器配置为以 Thumb 指令集为目标。

缺省设置

这是不支持 ARM 指令集的目标的缺省选项。

另请参阅

- 第2-8 页的*--arm*
- 第4-56 页的*#pragma arm*
- 第4-70 页的*#pragma thumb*
- *《编译器用户指南》*中第2-23 页的*指定目标处理器或体系结构*
- *《编译器用户指南》*中第5-3 页的*选择目标 CPU*。

2.1.152 --translate_g++

此选项通过启用 GNU 工具的命令行转换，有助于在 C++ 模式中仿真 GNU 编译器。

用法

您可以使用此选项来提供以下任意项：

- 以 ARM Linux 为目标的完整 GCC 仿真。
- 完整 GCC 仿真的子集，表现形式为将各个 GCC 命令行参数转换为其 RVCT 等效项。

要提供完整的 ARM Linux GCC 仿真，还必须使用 --arm_linux_config_file。此选项组合可选择配置文件指定的相应 GNU 头文件和库，并包括对某些缺省行为的更改。

要将 GCC 命令行参数转换为其 RVCT 等效项而不以完整的 GCC 仿真为目标，请使用 --translate_g++ 仿真 g++，但不要与 --arm_linux_config_file 一起使用。由于使用此方法时不以完整的 GCC 仿真为目标，因此将保留 RVCT 缺省行为，并且不以 ARM Linux 为目标设置任何缺省值。 RVCT 库路径和选项缺省值保持不变。

限制

如果在命令行中指定 ARM Linux 配置文件并使用 `--translate_g++`，则会改变 `--exceptions`、 `--no_exceptions`、`--bss_threshold`、`--relaxed_ref_def`、`--no_relaxed_ref_def`、`--signed_bitfields` 和 `--unsigned_bitfields` 缺省设置。

另请参阅

- 第2-8 页的*--arm_linux*
- 第2-10 页的*--arm_linux_config_file=path*
- 第2-11 页的*--arm_linux_configure*
- 第2-13 页的*--arm_linux_paths*
- 第2-20 页的*--bss_threshold=num*
- 第2-24 页的*--configure_cpp_headers=path*
- 第2-25 页的*--configure_extra_includes=paths*
- 第2-26 页的*--configure_extra_libraries=paths*
- 第2-27 页的*--configure_gcc=path*
- 第2-28 页的*--configure_gld=path*
- 第2-29 页的*--configure_sysroot=path*
- 第2-53 页的*--exceptions*, *--no_exceptions*
- 第2-107 页的*--relaxed_ref_def*, *--no_relaxed_ref_def*
- 第2-111 页的*--shared*
- 第2-112 页的*--signed_bitfields*, *--unsigned_bitfields*
- 第2-120 页的*--translate_gcc*
- 第2-121 页的*--translate_gld*
- *《链接器参考指南》*中第2-3 页的*--arm_linux*
- *《链接器参考指南》*中第2-33 页的*--library=name*
- *《链接器参考指南》*中第2-51 页的*--[no_]search_dynamic_libraries*
- 《应用程序注释 212 — 使用 RVCT 4.0 版以及 GNU 工具和库生成 Linux 应用程序》(*Application Note 212 - Building Linux applications using RVCT v4.0 and the GNU Tools and Libraries*)。

2.1.153 --translate_gcc

此选项通过启用 GNU 工具的命令行转换，有助于仿真 gcc。

用法

您可以使用此选项来提供以下任意项：

- 以 ARM Linux 为目标的完整 GCC 仿真
- 完整 GCC 仿真的子集，表现形式为将各个 GCC 命令行参数转换为其 RVCT 等效项。

要提供完整的 GCC 仿真，还必须使用 --arm_linux_config_file。此选项组合可选择配置文件指定的相应 GNU 头文件和库，并包括对某些缺省行为的更改。

要将各个 GCC 命令行参数转换为其 RVCT 等效项而不以完整的 GCC 仿真为目标，请使用 --translate_gcc 仿真 gcc，但不要与 --arm_linux_config_file 一起使用。由于使用此方法时不以完整的 GCC 仿真为目标，因此将保留 RVCT 缺省行为，并且不以 ARM Linux 为目标设置任何缺省值。 RVCT 库路径和选项缺省值保持不变。

限制

如果在命令行中指定 ARM Linux 配置文件并使用 --translate_gcc，则会改变 --bss_threshold、--relaxed_ref_def、 --no_relaxed_ref_def、--signed_bitfields 和 --unsigned_bitfields 的缺省设置。

另请参阅

- 第2-8 页的*--arm_linux*
- 第2-10 页的*--arm_linux_config_file=path*
- 第2-11 页的*--arm_linux_configure*
- 第2-13 页的*--arm_linux_paths*
- 第2-24 页的*--configure_cpp_headers=path*
- 第2-25 页的*--configure_extra_includes=paths*
- 第2-26 页的*--configure_extra_libraries=paths*
- 第2-27 页的*--configure_gcc=path*
- 第2-28 页的*--configure_gld=path*
- 第2-29 页的*--configure_sysroot=path*
- 第2-107 页的*--relaxed_ref_def*, *--no_relaxed_ref_def*

- 第2-111 页的*--shared*
- 第2-112 页的*--signed_bitfields*, *--unsigned_bitfields*
- 第2-118 页的*--translate_g++*
- *--translate_gld*
- *《链接器参考指南》*中第2-3 页的*--arm_linux*
- *《链接器参考指南》*中第2-33 页的*--library=name*
- *《链接器参考指南》*中第2-51 页的*--[no_]search_dynamic_libraries*
- 《应用程序注释 212 — 使用 RVCT 4.0 版以及 GNU 工具和库生成 Linux 应用程序》(*Application Note 212 - Building Linux applications using RVCT v4.0 and the GNU Tools and Libraries*)。

2.1.154 --translate_gld

此选项通过启用 GNU 工具的命令行转换，有助于仿真 GNU ld。

用法

您可以使用此选项来提供以下任意项：

- 以 ARM Linux 为目标的完整 GNU ld 仿真
- 完整 GNU ld 仿真的子集，表现形式为将各个 GNU ld 命令行参数转换为其 RVCT 等效项。

要提供完整的 GNU ld 仿真，还必须使用 --arm_linux_config_file。此选项组合可选择配置文件指定的相应 GNU 头文件和库，并包括对某些缺省行为的更改。

要将各个 GNU ld 命令行参数转换为其 RVCT 等效项而不以完整的 GNU ld 仿真为目标，请使用 --translate_gld 仿真 GNU ld，但不要与 --arm_linux_config_file 一起使用。由于使用此方法时不以完整的 GNU ld 仿真为目标，因此将保留 RVCT 缺省行为，并且不以 ARM Linux 为目标设置任何缺省值。 RVCT 库路径和选项缺省值保持不变。

注意

- 调用 armcc 时会使用 --translate_gld，就像它是 GNU 链接器一样。它仅用于直接涉及 GNU 链接器的现有生成脚本。
- 在 gcc 和 g++ 模式中， armcc 使用 --translate_gld 将自身报告为自己所用的链接器。例如 gcc -print-file-name=ld。

另请参阅

- 第2-8 页的*--arm_linux*
- 第2-10 页的*--arm_linux_config_file=path*
- 第2-11 页的*--arm_linux_configure*
- 第2-13 页的*--arm_linux_paths*
- 第2-24 页的*--configure_cpp_headers=path*
- 第2-25 页的*--configure_extra_includes=paths*
- 第2-26 页的*--configure_extra_libraries=paths*
- 第2-27 页的*--configure_gcc=path*
- 第2-28 页的*--configure_gld=path*
- 第2-29 页的*--configure_sysroot=path*
- 第2-111 页的*--shared*
- 第2-118 页的*--translate_g++*
- 第2-120 页的*--translate_gcc*
- *《链接器参考指南》*中第2-3 页的*--arm_linux*
- *《链接器参考指南》*中第2-33 页的*--library=name*
- *《链接器参考指南》*中第2-51 页的*--[no_]search_dynamic_libraries*
- 《应用程序注释 212 — 使用 RVCT 4.0 版以及 GNU 工具和库生成 Linux 应用程序》(*Application Note 212 - Building Linux applications using RVCT v4.0 and the GNU Tools and Libraries*)。

2.1.155 --trigraphs, --no_trigraphs

此选项启用和禁用三元组识别。

缺省设置

缺省选项为 --trigraphs （在 GNU 模式中缺省为 --no_trigraphs）。

另请参阅

- *ISO/IEC 9899:TC2*。

2.1.156 -U*name*

此选项删除宏 *name* 的任何初始定义。

宏 *name* 可为以下任一类型

- 预定义宏
- 使用 -D 选项指定的宏。

注意

并非所有的编译器预定义宏都可以取消定义。

语法

-U*name*

其中：

name 是要取消定义的宏的名称。

用法

指定 -U*name* 与将文本 #undef *name* 放在每个源文件的开头的作用相同。

限制

编译器按以下顺序定义和取消定义宏：

1. 编译器预定义的宏
2. 使用 -D*name* 显式定义的宏
3. 使用 -U*name* 显式取消定义的宏。

另请参阅

- 第2-21 页的*-C*
- 第2-35 页的*-Dname[(parm-list)][=def]*
- 第2-51 页的*-E*
- 第2-86 页的*-M*
- 第4-115 页的*编译器预定义*

2.1.157 --unaligned_access, --no_unaligned_access

此选项启用或禁用基于 ARM 体系结构的处理器上的未对齐数据访问。

缺省设置

对于支持未对齐数据访问的基于 ARM 体系结构的处理器，缺省为 --unaligned_access。这包括：

- 基于 ARMv6 体系结构的所有处理器
- 基于 ARMv7-A 和 ARMv7-R 体系结构的处理器。

对于不支持未对齐数据访问的基于 ARM 体系结构的处理器，缺省为 --no_unaligned_access。这包括：

- 基于 ARMv6 以前版本的体系结构的所有处理器
- 基于 ARMv7-M 体系结构的处理器。

用法

--unaligned_access

在支持未对齐数据访问的处理器（如 --cpu=ARM1136J-S）上使用 --unaligned_access 可加快对压缩结构的访问速度。

若要启用未对齐支持，必须执行下列操作：

- 在初始化代码中清除 CP15 寄存器 1 的 A 位（即位 1）。
- 在初始化代码中设置 CP15 寄存器 1 的 U 位（即位 22）。

  U 位的初始值由内核的 **UBITINIT** 输入确定。

RVCT 库包含旨在利用未对齐访问的某些库函数的特殊版本。在启用未对齐访问支持的情况下，RVCT 工具将使用这些库函数从未对齐访问中获益。

--no_unaligned_access

使用 --no_unaligned_access 可在 ARMv6 处理器上禁止生成未对齐字和半字访问。

若要在不使用未对齐访问的情况下在 ARMv6 目标上启用对四字节求模的对齐检查，必须执行下列操作：

- 在初始化代码中设置 CP15 寄存器 1 的 A 位（即位 1）。
- 在初始化代码中设置 CP15 寄存器 1 的 U 位（即位 22）。

  U 位的初始值由内核的 **UBITINIT** 输入确定。

注意

ARM 处理器内核不支持未对齐双字访问，例如对 **long long** 整数的未对齐访问。双字访问必须是八字节或四字节对齐的。

编译器不支持对八字节求模的对齐检查。也就是说，编译器（或更具体地说是 RVCT 工具集）不支持 CP15 寄存器 1 中的配置 U = 0、A = 1。

RVCT 库包含旨在利用未对齐访问的某些库函数的特殊版本。若要在禁用未对齐访问支持的情况下禁止使用这些高级库函数，则在编译 C 和 C++ 源文件以及汇编语言源文件组合的情况下，需要同时在编译器命令行和汇编器命令行中指定 --no_unaligned_access。

限制

仅当软件中的对齐支持选项与处理器内核中的对齐支持选项相匹配时，针对支持未对齐数据访问的处理器而编译的代码才能正确运行。

另请参阅

- 第2-30 页的*--cpu=name*
- *《汇编器指南》*中第3-2 页的*命令语法*
- *《开发指南》*中第2-12 页的*对齐支持*。

2.1.158 --use_pch=*filename*

此选项指示编译器在当前编译中包含具有指定文件名的 PCH 文件。

如果在相同的命令行中包括 --pch 和此选项，则后者优先。

语法

--use_pch=*filename*

其中：

filename 是要包含在当前编译中的 PCH 文件。

限制

如果在相同的命令行包括 --create_pch=*filename* 和此选项，则后者将不起作用。

错误

如果指定的文件不存在或者是无效的 PCH 文件，则编译器会生成错误。

另请参阅

- 第2-34 页的*--create_pch=filename*
- 第2-98 页的*--pch*
- 第2-98 页的*--pch_dir=dir*
- 第2-99 页的*--pch_messages*, *--no_pch_messages*
- 第2-99 页的*--pch_verbose*, *--no_pch_verbose*
- *《编译器用户指南》*中第2-18 页的*预编译的头文件*

2.1.159 --using_std, --no_using_std

当 C++ 中包含标准头文件时，此选项允许或禁止隐式使用 std 命名空间。

注意

此选项仅作为不符合 C++ 标准的旧式源代码的迁移辅助选项。不建议使用此选项。

模式

仅当源语言为 C++ 时，此选项才有效。

缺省设置

缺省选项为 --no_using_std。

另请参阅

- 第5-15 页的*命名空间*

2.1.160 --vectorize, --no_vectorize

使用此选项可允许或禁止直接从 C 或 C++ 代码生成 NEON 向量指令。

缺省设置

缺省为 --no_vectorize。

限制

必须为要向量化的循环指定以下选项：

`--cpu=name` 目标处理器必须具有 NEON 功能。

`-Otime` 缩短执行时间的优化类型。

`-Onum` 优化级别。必须使用以下选项之一：

- `-O2` 高度优化。这是缺省设置。
- `-O3` 最大优化。

注意

NEON 是 ARM 高级*单指令多数据* (SIMD) 扩展的实现。

若要启用向量化，需要单独的 *FLEXnet* 许可证。该许可证随 RVDS 4.0 Professional 一起提供。

示例

```
armcc --vectorize --cpu=Cortex-A8 -O3 -Otime -c file.c
```

另请参阅

- 第2-30 页的*--cpu=name*
- 第2-94 页的*-Onum*
- 第2-96 页的*-Otime*

2.1.161 --vfe, --no_vfe

此选项启用或禁用 C++ 中的*虚拟函数删除* (VFE)。

使用 VFE 可以将未使用的虚拟函数从代码中删除。启用 VFE 时，编译器会将信息放在带有前缀 `.arm_vfe_` 的特殊节中。不识别 VFE 的链接器将忽略这些节，因为其余代码部分不会引用这些节。因此，这些节不会增加可执行文件的大小，但会增加对象文件的大小。

模式

仅当源语言为 C++ 时，此选项才有效。

缺省设置

缺省为 --vfe （用 RVCT v2.1 以前版本的编译器编译的旧对象文件不包含 VFE 信息的情况除外）。

另请参阅

- 第C-3 页的*调用纯虚函数*
- *《链接器用户指南》*中第3-11 页的*未使用虚函数删除*。

2.1.162 --via=*filename*

此选项指示编译器从指定的文件中读取其他命令行选项。从文件中读取的选项将添加到当前命令行中。

Via 命令可嵌套在 via 文件中。

语法

--via=*filename*

其中：

filename 是 via 文件的名称，该文件包含要包括在命令行中的选项。

示例

假设有一个源文件 main.c、一个包含以下一行的 via 文件 apcs.txt：

```
--apcs=/rwpi --no_lower_rwpi --via=L_apcs.txt
```

以及另一个包含以下一行的 via 文件 L_apcs.txt：

```
-L--rwpi -L--callgraph
```

用以下命令行编译 main.c 时：

```
armcc main.c -L-o" main.axf" --via=apcs.txt
```

使用以下命令行编译 main.c：

```
armcc --no_lower_rwpi --apcs=/rwpi -L--rwpi -L--callgraph -L-o"main.axf" main.c
```

另请参阅

- 附录 A *via 文件语法*
- *《编译器用户指南》*中第2-12 页的*从文件读取编译器选项*。

2.1.163 --vla, --no_vla

此编译启用或禁用对可变长度数组的支持。

缺省设置

缺省情况下， C90 和标准 C++ 不支持可变长度数组。选择 --vla 选项可启用对 C90 或标准 C++ 中可变长度数组的支持。

标准 C 和 GNU 编译器扩展均支持可变长度数组。如果源语言是 C99 或者指定了 --gnu 选项，则将隐式选择 --vla 选项。

示例

```
size_t arr_size(int n)
{
    char array[n];          // variable length array, dynamically allocated
    return sizeof array;   // evaluated at runtime
}
```

另请参阅

- 第2-22 页的*--c90*
- 第2-22 页的*--c99*
- 第2-30 页的*--cpp*
- 第2-66 页的*--gnu*

2.1.164 --vsn

此选项显示版本信息和许可证详细信息。

另请参阅

- 第2-69 页的*--help*

2.1.165 -W

此选项指示编译器禁止显示所有警告消息。

另请参阅

- 第2-19 页的*--brief_diagnostics*, *--no_brief_diagnostics*
- 第2-44 页的*--diag_error=tag[,tag,...]*
- 第2-45 页的*--diag_remark=tag[,tag,...]*
- 第2-45 页的*--diag_style={arm|ide|gnu}*
- 第2-46 页的*--diag_suppress=tag[,tag,...]*
- 第2-48 页的*--diag_warning=tag[,tag,...]*
- 第2-52 页的*--errors=filename*
- 第2-108 页的*--remarks*
- 第2-133 页的*--wrap_diagnostics*, *--no_wrap_diagnostics*

2.1.166 --wchar, --no_wchar

此选项允许或禁止使用 wchar_t。只要这些选项未被使用，则不必是错误声明。

用法

使用此选项可创建与 wchar_t 大小无关的对象文件。

限制

如果指定了 --no_wchar：

- 编译器将把结构声明中的 wchar_t 域视为错误，而与是否使用该结构无关
- 编译器将 typedef 中的 wchar_t 视为错误，而与是否使用该 typedef 无关。

缺省设置

缺省选项为 --wchar。

另请参阅

- 第2-131 页的*--wchar16*
- 第2-131 页的*--wchar32*

2.1.167 --wchar16

此选项将 wchar_t 的类型更改为 **unsigned short**。

选择此选项会同时修改 C 中已定义类型 wchar_t 的类型以及 C++ 中本机类型 **wchar_t** 的类型。此外，还会影响 WCHAR_MIN 和 WCHAR_MAX 的值。

缺省设置

除非显式指定 --wchar32，否则编译器将采用 --wchar16。

另请参阅

- 第2-130 页的*--wchar, --no_wchar*
- *--wchar32*
- 第4-115 页的*预定义宏*

2.1.168 --wchar32

此选项将 wchar_t 的类型更改为 **unsigned int**。

选择此选项会同时修改 C 中已定义类型 wchar_t 的类型以及 C++ 中本机类型 **wchar_t** 的类型。此外，还会影响 WCHAR_MIN 和 WCHAR_MAX 的值。

缺省设置

除非显式指定 --wchar32 或在命令行中指定 ARM Linux 配置文件，否则编译器将采用 --wchar16。在命令行中指定 ARM Linux 配置文件将启用 --wchar32。

另请参阅

- 第2-10 页的*--arm_linux_config_file=path*
- 第2-11 页的*--arm_linux_configure*
- 第2-130 页的*--wchar, --no_wchar*
- *--wchar16*
- 第4-115 页的*预定义宏*

2.1.169 --whole_program

此选项向编译器保证整个程序由命令行中指定的源文件构成。这样，编译器将能够基于它看到的源代码就是要编译的程序的完整源代码集这一信息来应用优化。没有此信息，编译器将在对代码应用优化时更加保守。

用法

使用此选项可在编译小程序时获得最佳性能。

限制

如果您不能向编译器提供所有源代码，则不要使用此选项。

另请参阅

- 第2-90 页的*--multifile, --no_multifile*

2.1.170 --workdir=*directory*

此选项用于为项目模板提供工作目录。

注意

项目模板在包含文件 （如 RVD 配置文件）时仅需要工作目录。

语法

--workdir=*directory*

其中：

directory 是项目目录的名称。

限制

如果使用 --workdir 指定项目工作目录，则必须使用 --project 指定项目文件。

错误

如果要尝试使用不带 --workdir 的 --project，而 --workdir 又是必需的，则会生成错误消息。

另请参阅

- 第2-104 页的*--project=filename, --no_project=filename*
- 第2-107 页的*--reinitialize_workdir*

2.1.171 --wrap_diagnostics, --no_wrap_diagnostics

此选项允许或禁止在错误消息文本太长而在一行中放不下时，对错误消息文本进行换行。

缺省设置

缺省选项为 --no_wrap_diagnostics。

另请参阅

- 第2-19 页的*--brief_diagnostics, --no_brief_diagnostics*
- 第2-44 页的*--diag_error=tag[,tag,...]*
- 第2-45 页的*--diag_remark=tag[,tag,...]*
- 第2-45 页的*--diag_style={arm|ide|gnu}*
- 第2-46 页的*--diag_suppress=tag[,tag,...]*
- 第2-48 页的*--diag_warning=tag[,tag,...]*
- 第2-52 页的*--errors=filename*
- 第2-108 页的*--remarks*
- 第2-129 页的*-W*
- *《编译器用户指南》*中第 6 章 *诊断消息*。

第 3 章
语言扩展

本章介绍 ARM 编译器支持的语言扩展，包括：

- 第3-2 页的*预处理器扩展*
- 第3-4 页的*C90 中提供的 C99 语言功能*
- 第3-6 页的*C++ 和 C90 中提供的 C99 语言功能*
- 第3-9 页的*标准 C 语言扩展*
- 第3-14 页的*标准 C++ 语言扩展*
- 第3-18 页的*标准 C 和标准 C++ 语言扩展*
- 第3-23 页的*GNU 语言扩展*

有关 ARM 编译器的其他参考材料，另请参阅：

- 附录 B *标准 C 实现定义*
- 附录 C *标准 C++ 实现定义*
- 附录 D *C 和 C++ 编译器实现限制*

3.1 预处理器扩展

编译器支持几种预处理器扩展，包括 System V 版本 4 的 #assert 预处理扩展。

3.1.1 #assert

允许使用 System V 版本 4 的 #assert 预处理扩展。这些扩展可以定义和测试谓词名称。

此类名称位于与所有其他名称 （包括宏名称）不同的命名空间中。

语法

```
#assert name
#assert name[(token-sequence)]
```

其中：

name 是一个谓词名称

token-sequence 是一个可选的标记序列。
如果省略标记序列，则不会为 *name* 指定值。
如果包含标记序列，则会为 *name* 指定值 *token-sequence*。

示例

可以在 #if 表达式中测试使用 #assert 定义的谓词名称，例如：

```
#if #name(token-sequence)
```

如果具有标记序列 *token-sequence* 的名称 *name* 的 #assert 已出现，则此值为 1，否则为 0。可以在给定时间为给定谓词指定多个值。

另请参阅

- 第3-3 页的*#unassert*

3.1.2 #include_next

此预处理器指令是 #include 指令的变体。它仅在搜索路径中当前源文件 （即包含 #include_next 指令的文件）所在目录后面的目录中搜索已命名文件。

——注意——

此预处理程序指令是 ARM 编译器支持的 GNU 编译器扩展。

3.1.3 #unassert

可以使用 #unassert 预处理指令删除谓词名称。

语法

#unassert *name*

#unassert *name*[(*token-sequence*)]

其中：

name 是一个谓词名称

token-sequence 是一个可选的标记序列。

如果省略标记序列，则会删除 *name* 的所有定义。

如果包含标记序列，则只删除指示的定义。所有其他定义保持不变。

另请参阅

- 第3-2 页的*#assert*

3.1.4 #warning

支持预处理指令 #warning。与 #error 指令一样，此指令在编译时生成用户定义的警告。但不会暂停编译。

限制

如果指定 --strict 选项，则无法使用 #warning 指令。如果使用该指令，则会生成错误。

另请参阅

- 第2-116 页的*--strict*, *--no_strict*

3.2 C90 中提供的 C99 语言功能

编译器支持多种 ISO C90 标准扩展，例如， C99 样式的 // 注释。

如果源语言是 C90，并且是在非 strict 模式下进行编译，则可以使用这些扩展。

如果源语言是 C90，并且使用 --strict 编译器选项将编译器限制为编译严格 C90，则无法使用这些扩展。

注意

标准 C 和标准 C++ 的语言功能 （如 C++ 样式的 // 注释）可能与本节中介绍的 C90 语言扩展类似。如果使用 --strict 编译器选项编译严格标准 C 或严格标准 C++，则可以继续使用这些功能。

3.2.1 // 注释

字符序列 // 开始一行注释，就像在 C99 或 C++ 中一样。

C90 中的 // 注释与 C99 中的 // 注释具有相同的语义。

示例

```
// this is a comment
```

另请参阅

- 《编译器用户指南》中第5-43 页的*C99 的新功能*。

3.2.2 下标结构

在 C90 中，不是左值的数组仍会降级为指针，并且可以加下标。但是，在下一序列点后不能修改或使用它们，并且不能为其应用一元 & 运算符。可以在 C90 中为这种数组加下标，但它们在 C99 模式以外不会降级为指针。

示例

```
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 可变数组成员

struct 的最后一个成员可以具有不完整的数组类型。最后一个成员不能是 **struct** 的唯一成员，否则 **struct** 的大小为零。

示例

```
typedef struct
{
    int len;
    char p[]; // incomplete array type, for use in a malloced data structure
} str;
```

另请参阅

- 《编译器用户指南》中第5-43 页的*C99 的新功能*。

3.3 C++ 和 C90 中提供的 C99 语言功能

编译器支持多种 ISO C++ 标准和 C90 语言扩展，例如，覆盖旧式非原型定义的函数原型。

如果以下条件成立，则可以使用这些扩展：

- 源语言是 C++，并且在非 strict 模式下进行编译
- 源语言是 C90，并且在非 strict 模式下进行编译。

如果以下条件成立，则无法使用这些扩展：

- 源语言是 C++，并且使用 `--strict` 编译器选项将编译器限制为编译严格 C90。
- 源语言是 C90，并且使用 `--strict` 编译器选项将编译器限制为编译严格标准 C。

注意

标准 C 的语言功能 （如 `long long` 整数）可能与本节中介绍的语言扩展类似。如果使用 `--strict` 编译器选项编译严格标准 C++ 或严格 C90，则可以继续使用这些功能。

3.3.1 可变参数宏

在 C90 和 C++ 中，可以将宏声明为接受可变数量的自变量。

C90 和 C++ 中用于声明可变参数宏的语法遵循用于声明可变参数宏的 C99 语法，除非选择了 `--gnu` 选项。如果指定了 `--gnu` 选项，则该语法遵循可变参数宏的 GNU 语法。

示例

```
#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);
}
```

另请参阅

- 第2-66 页的*--gnu*
- 《编译器用户指南》中第5-43 页的*C99 的新功能*。

3.3.2 long long

ARM 编译器通过类型说明符 **long long** 和 **unsigned long long** 支持 64 位整型。就通常的算术转换而言，它们的行为类似于 **long** 和 **unsigned long**。__int64 是 **long long** 的同义词。

整型常数可以具有：

- ll 后缀，用于将常数类型转换为 **long long**（如果合适），或者转换为 **unsigned long long**（如果不合适）
- ull 或 llu 后缀，用于将常数的类型转换为 **unsigned long long**。

printf() 和 scanf() 的格式说明符可以包含 ll，以指定随后的转换应用于 **long long** 自变量，如在 %lld 或 %llu 中。

同样，如果普通整型常数的值足够大，则该常数的类型为 **long long** 或 **unsigned long long**。编译器将生成一条警告消息以指示这一变化。例如，在严格 1990 ISO 标准 C 中， 2147483648 的类型为 **unsigned long**。在 ARM C 和 C++ 中，其类型为 **long long**。这种情况产生的一个后果可以在表达式值中体现出来，例如：

```
2147483648 > -1
```

在严格 C 和 C++ 中，该表达式的计算结果为 0；而在 ARM C 和 C++ 中，该表达式的计算结果为 1。

long long 类型适用于通常的算术转换。

另请参阅

- 第4-9 页的*__int64*

3.3.3 restrict

restrict 关键字是一种 C99 功能，用于确保不同的对象指针类型和函数参数数组不会指向重叠的内存区域。因此，编译器可以执行优化，而不会由于可能的混淆而将优化禁止。

限制

要在 C90 或 C++ 中启用 **restrict** 关键字，必须指定 --restrict 选项。

支持将 __restrict 和 __restrict__ 关键字作为 **restrict** 的同义词，无论是否指定了 --restrict，始终可以使用这些关键字。

示例

```
void copy_array(int n, int *restrict a, int *restrict b)
{
    while (n-- > 0)
        *a++ = *b++;
}
```

另请参阅

- 第2-109 页的*--restrict, --no_restrict*
- 《编译器用户指南》中第5-43 页的*C99 的新功能*。

3.3.4 十六进制浮点

C90 和 C++ 支持可以按十六进制格式写入的浮点数。

示例

```
float hex_floats(void)
{
    return 0x1.fp3;    // 1.55e1
}
```

另请参阅

- 《编译器用户指南》中第5-43 页的*C99 的新功能*。

3.4 标准 C 语言扩展

编译器支持多种 ISO C99 标准扩展，例如，覆盖旧式非原型定义的函数原型。

如果以下条件成立，则可以使用这些扩展：

- 源语言是 C99，并且在非 strict 模式下进行编译
- 源语言是 C90，并且在非 strict 模式下进行编译。

如果以下条件成立，则无法使用其中的任何扩展：

- 源语言是 C90，并且使用 `--strict` 编译器选项将编译器限制为编译严格 C90。
- 源语言是 C99，并且使用 `--strict` 编译器选项将编译器限制为编译严格标准 C。
- 源语言是 C++。

3.4.1 常数表达式

初始值设定项中支持扩展的常数表达式。以下示例说明了缺省、`--strict_warnings` 和 `--strict` 编译器模式下的编译器行为。

示例 1，分配变量地址

代码可能包含在文件范围内分配变量地址的常数表达式，例如：

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

为 C 语言进行编译时，这会出现以下行为：

- 在缺省模式下，生成警告。
- 在 `--strict_warnings` 模式下，生成警告。
- 在 `--strict` 模式下，生成错误。

示例 2，常数初始值设定项

以下示例简要说明 C 代码中具有包含常数值的表达式时的编译器行为：

```
                                        /*  Std    RVCT v3.1  */
extern int  const c = 10;               /*  ok        ok      */
extern int  const x = c + 10;           /* 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     */
```

这指示由 ISO C 标准 Std 定义的行为以及 RVCT 中的行为：

- ok 表明在所有 C 模式下均接受该语句。
- ext 是 ISO C 标准的扩展。此行为取决于编译 C 时使用的 strict 模式：

  非严格

  接受，不会发出警告。

  --strict_warnings

  接受，但会发出警告。

  --strict

  符合 ISO C 标准，但会生成错误。

另请参阅

- 第2-55 页的*--extended_initializers, --no_extended_initializers*
- 第2-116 页的*--strict, --no_strict*
- 第2-117 页的*--strict_warnings*

3.4.2 数组和指针扩展

支持以下数组和指针扩展：

- 对于可互换但不完全相同的类型，允许指针之间存在分配和指针差别，例如，**unsigned char** * 和 **char** *。这包括指向相同大小的整型的指针，通常为 **int** * 和 **long** *。将发出警告。

  允许将字符串常数分配给指向任何类型字符的指针，而不会发出警告。

- 如果目标类型添加了非顶级的类型限定符，则允许分配指针类型；例如，将 **int** ** 分配给 **const int** **。这些指针类型对也允许存在比较和指针差别。将发出警告。
- 在指针运算中，如果需要，则始终将指向 void 的指针隐式转换为其他类型。同样，如果需要，始终将空指针常数隐式转换为正确类型的空指针。在 ISO C 中，某些运算符允许这样转换，而其他运算符不允许。

- 可以分配指向不同函数类型的指针，或者进行相等 (==) 或不相等 (!=) 比较，而无需进行显式类型转换。发出警告或错误。

  在 C++ 模式下，禁止使用该扩展。

- 可以将指向 **void** 的指针隐式转换为指向函数类型的指针，反之亦然。
- 在初始值设定项中，如果整型足够大以包含某个指针常数值，则可以将其转换为整型。
- 如果数组带有下标或以类似方式使用，则会将非左值的数组表达式转换为指向数组中的第一个元素的指针。

3.4.3 块范围函数声明

支持以下两个块范围函数声明扩展：

- 块范围函数声明还在文件范围内声明函数名称
- 块范围函数声明可能具有静态存储类，因而导致产生的声明缺省具有内部链接。

示例

```
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 标识符中的美元符号

标识符中允许使用美元符号 ($)。

——注意——

使用 --strict 选项进行编译时，可通过 --dollar 命令行选项允许在标识符中使用美元符号。

示例

```
#define DOLLAR$
```

另请参阅

- 第2-50 页的*--dollar, --no_dollar*
- 第2-116 页的*--strict, --no_strict*

3.4.5 顶级声明

C 输入文件可能不包含顶级声明。

错误

如果 C 输入文件不包含顶级声明，则会发出备注消息。

——注意——

缺省情况下不显示备注。要查看备注消息，请使用编译器选项 --remarks。

另请参阅

- 第2-108 页的*--remarks*

3.4.6 良性重声明

允许对 **typedef** 名称进行良性重声明。也就是说，可以在同一范围内将 **typedef** 名称重新声明为同一类型。

示例

```
typedef int INT;typedef int INT; /* redeclaration */
```

3.4.7 外部实体

在其他范围内声明的外部实体是可见的。

错误

如果在其他范围内声明的外部实体是可见的，编译器将生成警告。

示例

```
void f1(void)
{
    extern void f();
}
void f2(void)
{
    f(); /* Out of scope declaration */
}
```

3.4.8 函数原型

编译器可以识别覆盖在代码后面位置出现的旧式非原型定义的函数原型，例如：

错误

如果使用旧式函数原型，编译器将生成一条警告消息。

示例

```
int function_prototypes(char);
// Old-style function definition.
int function_prototypes(x)
    char x;
{
    return x == 0;
}
```

3.5 标准 C++ 语言扩展

编译器支持多种 ISO C++ 标准扩展，例如，类成员声明中的限定名称。

如果源语言是 C++，并且是在非 strict 模式下进行编译，则可以使用这些扩展。

如果源语言是 C++，并且使用 `--strict` 编译器选项将编译器限制为编译严格标准 C++，则无法使用这些扩展。

3.5.1 ? 运算符

如果 ? 运算符的第二个和第三个操作数为字符串文字或宽字符串文字，则可以将其隐式转换为 `char *` 或 `wchar_t *`。在 C++ 中，字符串文字是 `const`。可通过隐式转换将字符串文字转换为 `char *` 或 `wchar_t *`，从而删除 `const`。但是，这种转换仅适用于简单字符串文字。一种扩展是允许将其作为 ? 运算结果。

示例

```
char *p = x ? "abc" : "def";
```

3.5.2 类成员声明

可以在类成员声明中使用限定名称。

错误

如果在类成员声明中使用限定名称，则会发出警告。

示例

```
struct A
{
    int A::f();  // is the same as int f();
};
```

3.5.3 friend

class 的 **friend** 声明可以省略 class 关键字。

缺省情况下，不会在 **friend** 声明中执行访问检查。可以使用 `--strict` 命令行选项强制进行访问检查。

示例

```
class B;
class A
{
    friend B;  // is the same as "friend class B"
};
```

另请参阅

- 第2-116 页的*--strict, --no_strict*

3.5.4 读/写常数

外部常数的链接说明指示，常数可以动态初始化或具有可变成员。

注意

使用 "C++:read/write" 链接仅对通过 --apcs /rwpi 编译的代码是必需的。如果使用此选项重新编译现有代码，则必须更改动态初始化或具有可变成员的外部常数的链接说明。

使用 --apcs /rwpi 选项编译 C++ 不符合 ISO C++ 标准。示例 3-1 中的声明假定 x 在只读段中。

示例 3-1 外部访问

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

无法为常数动态初始化 x（包括用户定义的构造函数），并且 T 不能包含可变成员。示例 3-2中的新链接说明声明 x 位于读/写段中，即使使用常数对其初始化。允许动态初始化 x，并且 T 可以包含可变成员。其他文件中的 x、y 和 z 定义必须具有相同的链接说明。

示例 3-2 链接说明

```
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 */
    };
}
```

不能使用其他链接重新声明常数对象。示例 3-3 中的代码生成编译器错误。

示例 3-3 编译器错误

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

注意

由于 C 没有链接说明，因此，不能将 C++ 中声明的 **const** 对象用作 C 的 extern "C++:read/write"。

另请参阅

- 第2-4 页的*--apcs=qualifer...qualifier*

3.5.5 标量类型常数

可以在类内部定义标量类型的常数。这是一种旧格式，新格式使用初始化的静态数据成员。

错误

如果在类内部定义整型常数的成员，则会发出警告。

示例

```
class A
{
    const int size = 10; // must be static const int size = 10;
    int a[size];
};
```

3.5.6 非成员函数模板的特化

作为一种扩展，允许在非成员函数模板的特化上指定存储类。

3.5.7 类型转换

允许在指向 extern "C" 函数的指针和指向 extern "C++" 函数的指针之间进行类型转换。

示例

```
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 标准 C 和标准 C++ 语言扩展

编译器支持多种 ISO C99 和 ISO C++ 标准扩展，如各种整型扩展、各种浮点扩展、十六进制浮点常数以及匿名类、结构和联合。

如果以下条件成立，则可以使用这些扩展：

- 源语言是 C++，并且在非 strict 模式下进行编译
- 源语言是 C99，并且在非 strict 模式下进行编译
- 源语言是 C90，并且在非 strict 模式下进行编译。

如果以下条件成立，则无法使用这些扩展：

- 源语言是 C++，并且使用 `--strict` 编译器选项将编译器限制为编译严格 C++。
- 源语言是 C99，并且使用 `--strict` 编译器选项将编译器限制为编译严格标准 C。
- 源语言是 C90，并且使用 `--strict` 编译器选项将编译器限制为编译严格 C90。

3.6.1 寄存器变量地址

可以获取 **`register`** 存储类变量的地址。

错误

如果获取 **`register`** 存储类变量的地址，编译器将生成警告。

示例

```
void foo(void)
{
    register int i;
    int *j = &i;
}
```

3.6.2 函数自变量

可以为顶级函数声明以外的函数参数指定缺省自变量。例如，可以在 `typedef` 声明以及指向函数的指针和指向成员函数的指针声明中接受这些参数。

3.6.3 匿名类、结构和联合

支持将匿名类、结构和联合作为扩展。 C 和 C++ 中支持匿名结构和联合。

缺省情况下，可以在 C++ 中使用匿名联合。但是，如果要使用以下内容，则必须指定 anon_unions 编译指示：

- C 中的匿名联合和结构
- C++ 中的匿名类和结构。

可通过 **typedef** 名称将匿名联合引入到包含类中。与真正的匿名联合不同，不必直接对其进行声明。例如：

```
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.
};
```

扩展也可以实现匿名类和匿名结构，但前提是它们没有 C++ 特性。例如，不允许在匿名类和匿名结构中使用静态数据成员或成员函数、非公共成员以及嵌套类型 （匿名类、结构或联合除外）。例如：

```
#pragma anon_unions
struct A
{
    struct
    {
        int i, j;
    };                // Okay -- references to A::i and A::j are allowed.
};
```

另请参阅

- 第3-30 页的*未命名的字段*
- 第4-56 页的*#pragma anon_unions*, *#pragma no_anon_unions*

3.6.4 汇编器标签

汇编器标签指定用于 C 符号的汇编器名称。例如，可以让汇编器代码和 C 代码使用相同的符号名称，如 counter。因此，可以导出不同的名称供汇编器使用。

```
int counter __asm__("counter_v1") = 0;
```

这会导出符号 counter_v1，而不是导出符号 counter。

另请参阅

- 第4-5 页的*__asm*

3.6.5 空声明

允许使用空声明，即分号前面没有任何内容。

示例

```
; // do nothing
```

3.6.6 十六进制浮点常数

ARM 编译器实现了 C 中数字常数的语法扩展，以便将浮点常数显式指定为 IEEE 位模式。

语法

用于将浮点常数指定为 IEEE 位模式的语法是：

`0f_`*n* 将 8 位十六进制数 *n* 解释为 **float** 常数。必须恰好是 8 位。

`0d_`*nn* 将 16 位十六进制数 *nn* 解释为 **double** 常数。必须恰好是 16 位。

3.6.7 不完整的枚举

支持 enum 的正向声明。

示例

```
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 整型扩展

在整型常数表达式中，可以将整型常数转换为指针类型，然后再转换回整型。

3.6.9 标签定义

在标准 C 和标准 C++ 中，语句必须在标签定义后面。在 C 和 C++ 中，标签定义可以紧靠右花括号前面放置。

错误

如果标签定义紧靠右花括号前面放置，编译器将生成警告。

示例

```
void foo(char *p)
{
    if (p)
    {
        /* ... */
label:
    }
}
```

3.6.10 long float

long float 被视为 **double** 的同义词。

3.6.11 非静态局部变量

可以在非求值表达式中引用包含函数的非静态局部变量，例如，局部类中的 sizeof 表达式。将发出警告。

3.6.12 结构、联合、枚举和位域扩展

支持以下结构、联合、枚举和位域扩展：

- 在 C 中，文件范围数组的元素类型可以是不完整的 **struct** 或 **union** 类型。在需要使用元素类型大小之前，元素类型必须是完整的，例如，如果为数组加下标。如果数组不是 **extern**，则元素类型在编译结束时必须是完整的。
- 可以忽略 **struct** 或 **union** 说明符的右花括号 } 前面的最后一个分号。将发出警告。
- 不需要使用花括号将用于初始化整个静态数组 **struct** 或 **union** 的初始值设定项单值表达式括起来。 ISO C 要求使用花括号。
- 支持一种扩展以实现类似于 C++ 匿名联合的结构，其中包括以下内容：
  - 不仅允许使用匿名联合，而且还允许使用匿名结构。匿名结构的成员将升级为包含 **struct** 的范围，并且像普通成员一样对其进行查找。
  - 可通过 **typedef** 名称将其引入到包含 **struct** 中。即，不需要像真正匿名联合那样直接对其进行声明。
  - 可以声明标签，但只能在 C 模式下进行声明。

  要能够支持匿名结构和联合，必须使用 `anon_unions` 编译指示。
- 允许在 **enum** 列表末尾附加一个逗号，但会发出备注消息。
- **enum** 标签可以是不完整的。通过指定用花括号括起来的列表，可以定义标签名称并在以后对其进行解析。
- 可以由计算结果为无符号量的表达式给出枚举常数值，无符号量适用于 **unsigned int** 范围，而不适用于 **int** 范围。例如：

```
/* 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 */
```

- 位域可以具有基础类型，即 **enum** 类型或整型 （含 **int** 和 **unsigned int**）。

另请参阅

- 第4-55 页的*编译指示*
- *结构、联合、枚举和位域扩展*
- 《编译器用户指南》中第5-43 页的*C99 的新功能*。

3.7 GNU 语言扩展

本节介绍 ARM 编译器支持的 GNU 编译器扩展。仅在 GNU 模式下支持这些扩展，即，使用 `--gnu` 选项编译源代码时。有关详细信息，请参阅第 1-6 页的*语言遵从性*和第 2-66 页的*`--gnu`*。

注意

并非所有语言都支持各种 GNU 编译器扩展。例如， C++ 不支持扩展的指针算法。

有关使用 GNU 扩展的详细信息，请参阅 `http://gcc.gnu.org` 上的在线 GNU 编译器文档。

有关 ARM 编译器的其他参考材料，另请参阅：

- 附录 B *标准 C 实现定义*
- 附录 C *标准 C++ 实现定义*
- 附录 D *C 和 C++ 编译器实现限制*

3.7.1 替代关键字

编译器可识别 *`__keyword__`* 形式的替代关键字。这些替代关键字具有与原始关键字相同的行为。

示例

```
__const__ int pi = 3.14; // same as const int pi = 3.14
```

3.7.2 asm 关键字

此关键字是 `__asm` 关键字的同义词。

模式

仅在 C90 和 C99 的 GNU 模式下支持。

另请参阅

- 第 4-5 页的*`__asm`*

3.7.3 条件范围

可以在 **switch** 语句中指定值范围。

示例

```
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 union 类型转换

到 union 类型的转换类似于其他类型转换，只不过指定的类型是 union 类型。可以使用 union 标签或 typedef 名称指定类型。

模式

仅在 C90 和 C99 的 GNU 模式下支持。

示例

```
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 字符转义序列

在字符串中，转义序列'\e'被视为转义字符 <ESC> (ASCII 27)。

示例

```
void foo(void)
{
    printf("Escape sequence is: \e\n");
}
```

3.7.6 复合文字

与在 C99 中一样，支持复合文字。所有复合文字都是左值。

示例

```
int y[] = (int []) {1, 2, 3}; // error in strict C99, okay in C99 --gnu
int z[] = (int [3]) {1};
```

模式

仅在 C90 和 C99 的 GNU 模式下支持。

注意

也可以在 C99 中将复合文字用作初始值设定项。但是，编译器对在 GNU 模式下视为初始值设定项的复合文字的要求比编译 C99 源代码时的要求宽松一些。

3.7.7 条件语句

如果结果与测试相同，则条件语句中的中间操作数可以忽略。

例如：

示例

以下语句是等效的：

```
c = i ? : j; // middle operand omitted
c = i ? i : j;
```

```
if (i) c = i; else c = j; // expanded in full
```

如果测试以某种方式修改了值，这是非常有用的，例如：

```
i++ ? : j;
```

其中，i++ 来自宏。如果以这种方式编写代码，则只计算 i++ 一次。

如果 i 的原始值不为零，则结果为 i 的原始值。无论此结果如何，i 都会增加一次。

模式

仅在 GNU 模式下支持。支持的语言为 C90、 C99 和 C++。

3.7.8 指定的初始值设定项

与在 C99 中一样，支持指定的初始值设定项。

示例

```
int a[6] = { [4] = 29, [2] = 15 };
int b[6] = { 0,0,15,0,29,0 }; // a[] is equivalent to b[]
```

模式

仅在 C90 和 C++ 的 GNU 模式下支持。

另请参阅

- 《编译器用户指南》中第5-43 页的*C99 的新功能*。

3.7.9 扩展的左值

在 GNU 模式下，在查看逗号表达式和 ?: 结构时对构成左值的内容的定义较为宽松。可以使用复合表达式、条件表达式和类型转换，如下所示：

- 可以指定复合表达式：

  ```
  (a++, b) += x;
  ```

  这相当于：

  ```
  temp = (a++,b);
  b = temp + x
  ```

- 可以获取复合表达式 &(a, b) 的地址。这相当于 (a, &b)。

- 可以使用条件表达式，例如：

  ```
  (a ? b : c) = a;
  ```

  这会选取 b 或 c 作为目标，具体取决于 a。

模式

仅在 C90 和 C99 的 GNU 模式下支持。

3.7.10 初始值设定项

与在标准 C++ 和 ISO C99 中一样，自动变量的聚合初始值设定项元素不需要是常数表达式。

模式

仅在 C90 的 GNU 模式下支持。

示例

```
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;
}
```

3.7.11 内联

inline 函数限定符是用于编译器的提示，表明将内联该函数。

`static inline foo (){...}`

foo 用于文件内部，并且不会导出符号。

`inline foo(){...}`

foo 用于文件内部、提供外联版本并导出名称 foo。

`extern inline foo (){...}`

在 GNU 模式下，如果已内联，则在内部使用 foo。如果没有内联，则引用外部版本而不是使用内部版本调用。同样地，不会发出 foo 符号。

在非 GNU 模式下，忽略 **extern**，并且功能与 C++ 的 **inline** foo() 相同。在 C 中，必须使用 __inline。有关详细信息，请参阅第5-18 页的*外部内联函数*。

模式

仅在 C90 的 GNU 模式下支持。

3.7.12 标签用作值

编译器支持使用 && 运算符将 GCC 用作值。

模式

在 C 和 C++ 的 GNU 模式下支持。

示例

标签表：

```
int f(int n)
{
    void *const table[] = { &&a1, &&a2};
    goto *table[n];
a1: return 1;
a2: return 2;
}
```

用于表示继续的标签：

```
void *toggle(void *lab, int *x)
{
    if (lab) goto *lab;
a1: *x = 1; return &&a2;
a2: *x = 0; return &&a1;
}
```

3.7.13 指针算法

可以执行有关 void 指针和函数指针的算法。

void 类型或函数类型的大小被定义为 1。

模式

仅在 C90 和 C99 的 GNU 模式下支持。

错误

如果编译器检测到有关 **void** 指针或函数指针的算法，则会生成警告。

示例

```
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;
}
```

3.7.14 语句表达式

语句表达式允许将包括声明在内的整段代码放在花括号 ({ }) 中。

语句表达式的结果是语句列表中的最后一项。

限制

不允许使用以语句表达式为目标的跳转。

在 C++ 模式中，还不允许跳转出去。在语句表达式中，不允许使用变长数组、可析构的实体、 try 语句、捕获、局部非 POD 类定义以及动态初始化的局部静态变量。

示例

```
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;
}
```

3.7.15 未命名的字段

将结构或联合嵌入到另一个结构或联合中时，不需要命名内部结构。可以访问未命名结构的内容，而无需使用 .name 引用它。

未命名的字段与匿名联合和结构相同。

模式

仅在 C90 和 C99 的 GNU 模式下支持。

示例

```
struct
{
    int a;
    union
    {
        int b;
        float c;
    };
    int d;
} Unnamed_Fields_0;
int Unnamed_Fields_1()
{
    return Unnamed_Fields_0.b;
}
```

另请参阅

- 第3-19 页的*匿名类、结构和联合*

第 4 章
编译器特有的功能

本章介绍 ARM 编译器特有的功能，包括：

4.1 关键字和运算符

本节介绍 ARM 编译器 armcc 支持的函数关键字和运算符。

表 4-1 列出的关键字是 ARM 对 C 和 C++ 标准的扩展。表中没有介绍不具备 ARM 编译器特有的行为或限制的标准 C 和 C++ 关键字。

表 4-1 ARM 编译器支持的关键字扩展

关键字		
__align	__int64	__svc
__ALIGNOF__	__INTADDR__	__svc_indirect
__asm	__irq	__svc_indirect_r7
__declspec	__packed	__value_in_regs
__forceinline	__pure	__weak
__global_reg	__softfp	__writeonly
__inline	__smc	

4.1.1 __align

__align 关键字指示编译器在 *n* 字节边界上对齐变量。

__align 是一个存储类修饰符。它不影响函数的类型。

语法

__align(*n*)

其中：

n 是对齐边界。

对于局部变量，*n* 值可为 1、 2、 4 或 8。

对于全局变量，*n* 可以具有最大为 2 的 0x80000000 次幂的任何值。

__align 关键字紧靠变量名称前面放置。

用法

如果声明的变量的常规对齐边界小于 *n*，__align(*n*) 是非常有用的。八字节对齐方式可以显著提高 VFP 指令的性能。

可以将 __align 与 **extern** 和 **static** 一起使用。

限制

由于 __align 是存储类修饰符，因此不能将其用于：

- 类型，包括 **typedef** 和结构定义
- 函数参数。

只能进行过对齐。也就是说，可以将两个字节的对象按 4 个字节对齐，而不能将 4 个字节的对象按两个字节对齐。

示例

```
__align(8) char buffer[128];  // buffer starts on eight-byte boundary

void foo(void)
{
    ...
    __align(16) int i; // this alignment value is not permitted for
                       // a local variable
    ...
}

__align(16) int i; // permitted as a global variable.
```

另请参阅

- *《编译器用户指南》*中第2-89 页的*--min_array_alignment=opt*

4.1.2 __alignof__

可以使用 __alignof__ 关键字查询有关类型或变量的对齐信息。

注意

此关键字是 ARM 编译器支持的 GNU 编译器扩展。

语法

__alignof__(*type*)

```
__alignof__(expr)
```

其中：

type 是类型

expr 是左值。

返回值

__alignof__(*type*) 返回类型 *type* 的对齐要求；如果没有对齐要求，则返回 1。

__alignof__(*expr*) 返回左值 *expr* 的类型的对齐要求；如果没有对齐要求，则返回 1。

示例

```
int Alignment_0(void)
{
    return __alignof__(int);
}
```

另请参阅

- *__ALIGNOF__*

4.1.3 __ALIGNOF__

__ALIGNOF__ 关键字返回指定类型的对齐要求，或者返回指定对象的类型的对齐要求。

语法

```
__ALIGNOF__(type)
```

```
__ALIGNOF__(expr)
```

其中：

type 是类型

expr 是左值。

返回值

__ALIGNOF__(*type*) 返回类型 *type* 的对齐要求；如果没有对齐要求，则返回 1。

__ALIGNOF__(*expr*) 返回左值 *expr* 的类型的对齐要求；如果没有对齐要求，则返回 1。不会计算左值本身的结果。

示例

```
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
```

另请参阅

- 第4-3 页的*__alignof__*

4.1.4 __asm

此关键字用于将信息从编译器传递到 ARM 汇编器 armasm。

此关键字执行的精确操作取决于其用法。

用法

嵌入式汇编器

可以使用 __asm 关键字声明或定义嵌入式汇编函数。例如：

```
__asm void my_strcpy(const char *src, char *dst);
```

有关详细信息，请参阅 *《编译器用户指南》*中第7-16 页的*嵌入式汇编器*。

内联汇编器

可以使用 __asm 关键字将内联汇编合并到函数中。例如：

```
int qadd(int i, int j)
{
    int res;
    __asm
    {
        QADD   res, i, j
    }
    return res;
}
```

有关详细信息，请参阅 *《编译器用户指南》*中第7-2 页的*内联汇编器*。

汇编器标签

可以使用 __asm 关键字为 C 符号指定汇编器标签。例如：

```
int count __asm__("count_v1"); // export count_v1, not count
```

有关详细信息，请参阅第3-20 页的*汇编器标签*。

已命名的寄存器变量

可以使用 __asm 关键字声明已命名的寄存器变量。例如：

```
register int foo __asm("r0");
```

有关详细信息，请参阅第4-108 页的*已命名的寄存器变量*。

另请参阅

- 第3-23 页的*asm 关键字*

4.1.5 __forceinline

__forceinline 关键字强制编译器内联编译 C 或 C++ 函数。

__forceinline 的语义与 C++ **inline** 关键字的语义完全相同。编译器尝试内联限定为 __forceinline 的函数，而不考虑其特性。但是，如果这样做导致出现问题，编译器将不内联函数。例如，递归函数仅内联到本身一次。

__forceinline 是一个存储类限定符。它不影响函数的类型。

——注意——

此关键字具有等效的函数属性 __attribute__((always_inline))。

示例

```
__forceinline static int max(int x, in y)
{
    return x > y ? x : y; // always inline if possible
}
```

另请参阅

- 第2-58 页的*--forceinline*
- 第4-33 页的*__attribute__((always_inline))*

4.1.6 __global_reg

__global_reg 存储类说明符将声明的变量分配给全局变量寄存器。

语法

__global_reg(*n*) *type varName*

其中：

n 是 1 到 8 之间的整数。

type 是以下类型之一：

- 除 **long long** 之外的任何整型
- 任何字符类型
- 任何指针类型。

varName 是变量名称。

限制

如果使用此存储类，则无法使用任何其他存储类，如 **extern**、**static** 或 **typedef**。

在 C 中，不能在声明时限定或初始化全局寄存器变量。在 C++ 中，任何初始化均被视为动态初始化。

可用寄存器的数量因所使用的 AAPCS 变体而异，有 5 到 7 个寄存器可用作全局变量寄存器。

实际上，使用全局寄存器变量时，建议：

- ARM 或 Thumb-2 中不要超过三个全局寄存器变量
- Thumb-1 中不要超过一个全局寄存器变量
- 全局浮点寄存器变量不要超过可用浮点寄存器数量的一半。

如果声明的全局变量太多，代码大小会显著增加。在某些情况下，程序可能无法进行编译。

小心

在使用全局寄存器变量时必须小心，原因如下：

- 在链接时不进行检查，因而无法保证不同编译单元之间的直接调用是合理的。如果可能，请在程序的每个编译单元内定义程序中使用的全局寄存器变量。通常，最好将定义放在全局头文件中。在使用全局寄存器之前，必须及早在代码中设置寄存器内的值。
- 全局寄存器变量映射到由被调用方保存的寄存器，因此，对于未将该变量用作全局寄存器变量的编译单元，将通过函数调用来保存和恢复它的值，例如，库函数。
- 回调使用全局寄存器变量的编译单元是很危险的。例如，如果从未声明某个全局寄存器变量的编译单元中调用使用该全局寄存器的函数，函数将从其假定的全局寄存器变量中读取错误的值。
- 只能在文件范围内使用此存储类。

示例

示例 4-1 声明一个分配给 r5 的全局变量寄存器。

示例 4-1 声明全局整数寄存器变量

```
__global_reg(2) int x; v2 is the synonym for r5
```

示例 4-2 将生成错误，因为必须在同一变量的所有声明中指定全局寄存器。

示例 4-2 全局寄存器 - 声明错误

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

在 C 中，无法在定义时初始化 __global_reg 变量。第4-9 页的示例 4-3 在 C 中生成错误，而在 C++ 中不生成错误。

示例 4-3 全局寄存器 - 初始化错误

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

另请参阅

- 第2-66 页的*--global_reg=reg_name[,reg_name,...]*

4.1.7 __inline

__inline 关键字提示编译器在合理的情况下内联编译 C 或 C++ 函数。

__inline 的语义与 **inline** 关键字的语义完全相同。但 C90 中不提供 **inline**。

__inline 是一个存储类限定符。它不影响函数的类型。

示例

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

另请参阅

- *《编译器用户指南》*中第5-16 页的*函数内联*。

4.1.8 __int64

__int64 关键字是关键字序列 **long long** 的同义词。

即使在使用 --strict 时，也接受 __int64。

另请参阅

- 第2-116 页的*--strict*, *--no_strict*
- 第3-7 页的*long long*

4.1.9 __INTADDR__

__INTADDR__ 运算将包含的表达式作为常数表达式处理，并将其转换为整型常数。

注意

它用于 offsetof 宏。

语法

__INTADDR(*expr*)

其中：

expr 是整型常数表达式。

返回值

__INTADDR__(*expr*) 返回一个与 *expr* 等效的整型常数。

另请参阅

- 《编译器用户指南》中第7-18 页的*嵌入式汇编程序的限制*。

4.1.10 __irq

通过使用 __irq 关键字，可以将 C 或 C++ 函数用作中断例程。

__irq 是一个函数限定符。它影响函数的类型。

限制

将保留所有损坏的寄存器 （浮点寄存器除外），而不仅限于通常在 AAPCS 中保留的寄存器。必须使用缺省 AAPCS 模式。

通过将程序计数器设置为 lr-4 并将 CPSR 设置为 SPSR 中的值，可以退出该函数。__irq 函数不能使用任何参数或返回值。

注意

针对纯 Thumb 指令模式的处理器进行编译时，代码将被编译为 Thumb 代码，因为在 Thumb 状态中进入中断处理程序。否则，即使使用 --thumb 选项或 #pragma thumb 来编译 Thumb 指令，指定为 __irq 的任何函数都将被编译为 ARM 代码。

另请参阅

- 第2-119 页的*--thumb*
- 第4-70 页的*#pragma thumb*
- *《开发指南》*中的第 6 章 *处理处理器异常*。

4.1.11 __packed

__packed 限定符将所有有效类型的对齐边界设置为 1。这就意味着：

- 不会插入填充以对齐压缩对象
- 使用未对齐的访问读取或写入压缩类型的对象。

使用 __packed 限定符声明结构或联合后，__packed 将应用于该结构或联合的所有成员。成员之间或结构末尾均没有填充。必须使用 __packed 声明压缩结构的所有子结构。可以单独压缩非压缩结构的整型子字段。

用法

若要将结构映射到外部数据结构或访问未对齐数据，__packed 限定符非常有用；但由于访问开销相对较高，通常对节省数据大小并没有什么帮助。通过仅对需要压缩的结构中的字段进行压缩，可以减少未对齐访问的数量。

——注意——

在硬件中不支持未对齐访问的 ARM 处理器 （例如， ARMv6 之前的处理器）上，访问未对齐的数据时可能会在代码大小和执行速度方面产生较高的成本。必须最大限度减少通过压缩结构进行的数据访问，以避免增加代码大小和降低性能。

限制

以下限制适用于使用 __packed 的场合：

- __packed 限定符不能用于以前未使用 __packed 声明的结构。
- 与其他类型限定符不同，不能同时具有同一结构类型的 __packed 版本和非 __packed 版本。
- __packed 限定符不影响整型局部变量。
- 压缩结构或联合与相应的非压缩结构的分配不兼容。由于这些结构具有不同的内存布局，因此，将压缩结构分配给非压缩结构的唯一办法是逐个字段进行复制。

- 没有定义对 __packed 进行类型转换所产生的影响。也没有定义将非压缩结构类型转换为压缩结构类型所产生的影响。可以合法地将指向整型的指针类型显式或隐式转换为指向压缩整型的指针类型。也可以对 **char** 类型进行 __packed 类型转换。
- 不存在压缩数组类型。压缩数组是指具有压缩类型的对象数组。数组中没有进行填充。

示例

示例 4-4 说明指针可以指向压缩类型。

示例 4-4 指向压缩类型的指针

```
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  */
                                    /* 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' */
```

示例 4-5 说明了使用指针访问压缩对象时，编译器可生成有效代码，并且代码与指针对齐方式无关。

示例 4-5 压缩结构

```
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
}
```

另请参阅

- 第4-50 页的*__attribute__((packed))*
- 第4-65 页的*#pragma pack(n)*
- 第5-10 页的*压缩结构*
- *《编译器用户指南》*中第5-25 页的*__packed 限定符和未对齐的数据访问*
- *《编译器用户指南》*中第5-27 页的*__packed 结构与单个 __packed 字段*

4.1.12 __pure

__pure 关键字指明声明的函数为纯函数。

只有在以下情况下，函数才是*纯函数*：
- 结果仅取决于其自变量值
- 函数没有副作用。

__pure 是一个函数限定符。它影响函数的类型。

注意

此关键字具有等效的函数属性 __attribute__((const))。

缺省设置

缺省情况下，假定函数不是纯函数。

用法

可以选择纯函数来删除公共子表达式。

限制

声明为纯函数的函数可能没有副作用。例如，纯函数：

- 无法调用非纯函数
- 无法使用全局变量或解除引用的指针，因为编译器假定函数无法访问内存（堆栈内存除外）
- 使用相同参数两次调用纯函数时，每次必须返回相同的值。

示例

```
int factr(int n) __pure
{
    int f = 1;
    while (n > 0)
        f *= n--;
    return f;}
```

另请参阅

- 第4-33 页的*__attribute__((const))*
- *《编译器用户指南》*中第5-13 页的*__pure*
- *《编译器用户指南》*中第5-14 页的*放置ARM 函数限定符*

4.1.13 __smc

__smc 关键字声明 SMC （*安全监控调用*）函数。 SMC 函数调用在编译器生成的指令流中插入一个 SMC 指令 （在函数调用位置）。

注意

SMC 指令替代以前版本的 ARM 汇编语言中使用的 SMI 指令。

__smc 是一个函数限定符。它影响函数的类型。

语法

```
__smc(int smc_num) return-type function-name([argument-list]);
```

其中：

smc_num 是 SMC 指令中使用的 4 位立即值。

ARM 处理器忽略 *smc_num* 值，但 SMC 异常处理程序可以使用它来确定所请求的服务。

限制

如果所选的基于 ARM 体系结构的处理器具有安全扩展，则它们可以使用 SMC 指令。有关详细信息，请参阅 *《汇编器指南》*中第4-136 页的*SMC*。

如果针对不支持 SMC 指令的体系结构编译包含 __smc 关键字的源代码，编译器将生成错误。

示例

```
__smc(5) void mycall(void); /* declare a name by which SMC #5 can be called */
...
mycall();                   /* invoke the function */
```

另请参阅

- *《汇编器指南》*中第4-136 页的*SMC*。

4.1.14 __softfp

__softfp 关键字指明函数使用软件浮点链接。

__softfp 是一个函数限定符。它影响函数的类型。

注意

此关键字具有等效的 #pragma #pragma __softfp_linkage。

用法

对该函数的调用将在整数寄存器中传递浮点自变量。如果结果是浮点值，则在整数寄存器中返回该值。这与针对软件浮点的编译行为是相同的。

该关键字允许编译的源代码使用相同的库，以便使用硬件和软件浮点。

——注意——

在 C++ 中，如果要覆盖使用 __softfp 关键字限定的虚拟函数，还必须将覆盖函数声明为 __softfp。如果这些函数不匹配，编译器将生成错误。

另请参阅

- 第2-61 页的*--fpu=name*
- 第4-67 页的*#pragma softfp_linkage*, *#pragma no_softfp_linkage*
- *《编译器用户指南》*中第5-35 页的*浮点计算和链接*。

4.1.15 __svc

__svc 关键字声明*超级用户调用* (SVC) 函数，该函数最多使用四个类似于整数的参数，并通过 value_in_regs 结构最多返回四个结果。

__svc 是一个函数限定符。它影响函数的类型。

语法

```
__svc(int svc_num) return-type function-name([argument-list]);
```

其中：

svc_num
是在 SVC 指令中使用的立即值。
它是一个表达式，其计算结果为以下范围内的整数：
- 在 ARM 指令中为 0 到 $2^{24} - 1$ （24 位值）
- 在 16 位 Thumb 指令中为 0-255 （8 位值）。

用法

这导致将函数调用作为与 AAPCS 兼容的运算进行内联编译，此运算的行为与普通函数调用类似。

__value_in_regs 限定符可用于指定在寄存器中最多返回 16 个字节的小型结构，而不是由 AAPCS 中定义的常用结构传递机制返回。

示例

```
__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
```

错误

在命令行中使用 --cpu 选项指定不支持 SVC 指令的 ARM 体系结构版本或基于 ARM 体系结构的处理器时，编译器将生成错误。

另请参阅

- 第2-30 页的*--cpu=name*
- 第4-19 页的*__value_in_regs*
- *《汇编器指南》*中第4-129 页的*SVC*。

4.1.16 __svc_indirect

__svc_indirect 关键字在 r12 中向 SVC 处理程序传递操作码。

__svc_indirect 是一个函数限定符。它影响函数的类型。

语法

```
__svc_indirect(int svc_num)
        return-type function-name(int real_num[, argument-list]);
```

其中：

svc_num

是在 SVC 指令中使用的立即值。

它是一个表达式，其计算结果为以下范围内的整数：

- 在 ARM 指令中为 0 到 $2^{24} - 1$ （24 位值）
- 在 16 位 Thumb 指令中为 0-255 （8 位值）。

real_num 是在 r12 中传递给处理程序的值，用于确定要执行的函数。

要使用间接机制，系统处理程序必须使用 r12 值选择所需的运算。

用法

可以使用此功能实现间接 SVC。

示例

```
int __svc_indirect(0) ioctl(int svcino, int fn, void *argp);
```

调用：

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

可编译为 SVC #0， r12 中为 IOCTL+4。

错误

在命令行中使用 --cpu 选项指定不支持 SVC 指令的 ARM 体系结构版本或基于 ARM 体系结构的处理器时，编译器将生成错误。

另请参阅

- 第2-30 页的*--cpu=name*
- 第4-19 页的*__value_in_regs*
- *《汇编器指南》*中第4-129 页的*SVC*。

4.1.17 __svc_indirect_r7

__svc_indirect_r7 关键字的行为与 __svc_indirect 类似，但它使用的是 r7 而不是 r12。

__svc_indirect_r7 是一个函数限定符。它影响函数的类型。

语法

```
__svc_indirect_r7(int svc_num)
        return-type function-name(int real_num[, argument-list]);
```

其中：

svc_num 是在 SVC 指令中使用的立即值。

它是一个表达式，其计算结果为以下范围内的整数：

- 在 ARM 指令中为 0 到 $2^{24} - 1$ （24 位值）
- 在 16 位 Thumb 指令中为 0-255 （8 位值）。

real_num 是在 r7 中传递给处理程序的值，用于确定要执行的函数。

用法

ARM Linux 上的 Thumb 应用程序使用 __svc_indirect_r7 进行内核系统调用。

也可以使用此功能实现间接 SVC。

示例

```
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))
```

调用：

```
write(fd, buf, count);
```

可编译为 SVC #0， r0 = fd、 r1 = buf、 r2 = count 以及 r7 = 4。

错误

在命令行中使用 --cpu 选项指定不支持 SVC 指令的 ARM 体系结构版本或基于 ARM 体系结构的处理器时，编译器将生成错误。

另请参阅

- 第4-19 页的*__value_in_regs*
- 第2-30 页的*--cpu=name*
- *《汇编器指南》*中第4-129 页的*SVC*。

4.1.18 __value_in_regs

__value_in_regs 限定符指示编译器在整数寄存器中最多返回四个整型字的结构，或者在浮点寄存器中最多返回四个浮点或双精度值，而不是使用内存。

__value_in_regs 是一个函数限定符。它影响函数的类型。

语法

```
__value_in_regs return-type function-name([argument-list]);
```

其中：

return-type　　是大小最多为四个字的结构类型。

用法

调用返回多个结果的函数时，使用 __value_in_regs 声明函数是非常有用的。

限制

如果 __value_in_regs 结构需要复制构造， C++ 函数将无法返回该结构。

如果要覆盖声明为 __value_in_regs 的虚拟函数，还必须将覆盖函数声明为 __value_in_regs。如果这些函数不匹配，编译器将生成错误。

错误

如果由 __value_in_regs 限定的函数中返回的结构太大，则会生成警告并忽略 __value_in_regs 结构。

示例

```
typedef struct int64_struct
{
    unsigned int lo;
    unsigned int hi;
} int64_struct;
__value_in_regs extern
    int64_struct mul64(unsigned a, unsigned b);
```

另请参阅

- *《编译器用户指南》*中第5-12 页的*__value_in_regs*。

4.1.19 __weak

此关键字指示编译器弱导出符号。

可以将 __weak 关键字应用于函数和变量声明以及函数定义。

用法

函数和变量声明

对于声明，此存储类指定一个 **extern** 对象声明，即使不存在，也不会导致链接器将未解析的引用作为错误处理。

例如：

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

如果从编译为跳转或跳转链接指令的代码中对缺少的弱函数进行引用，则会：

- 将该引用解析为下一条指令的跳转。这实际上将跳转变为 NOP。
- 将该跳转替换为 NOP 指令。

函数定义

使用 __weak 定义的函数将弱导出其符号。除非将相同名称的非弱定义函数链接到相同映像上，否则弱定义函数的行为与正常定义的函数类似。如果非弱定义函数和弱定义函数位于相同映像中，则会将对弱定义函数的所有调用解析为对非弱函数的调用。如果有多个可用的弱定义，链接器将选择其中的一个弱定义供所有调用使用。

如果使用 __weak 声明函数，但随后没有使用 __weak 对其进行定义，则此函数与非弱函数的行为相同。

限制

使用 __weak 限定函数和变量声明以及函数定义时，存在一些限制。

函数和变量声明

在同一编译中，不能既弱使用又非弱使用函数或变量。例如，以下代码从 g() 和 h() 中弱使用 f()：

```
void f(void);
void g()
{
```

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

无法从定义某个函数或变量的同一编译中弱使用该函数或变量。以下代码从 h() 中非弱使用 f()：

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

链接器不会从库中加载函数或变量，除非其他编译非弱使用该函数或变量。如果一直没有解析引用，则假定其值为 NULL。但是，如果引用是指从代码中对位置无关节或缺少的 __weak 函数的引用，则未解析的引用不是 NULL。

函数定义

无法内联弱定义的函数。

示例

```
__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 /rwpi
__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
```

另请参阅

- *《实用程序指南》*中的第 3 章 *使用 armar*，以了解有关库搜索的详细信息。

4.1.20 __writeonly

__writeonly 类型限定符指示不能从中读取数据对象。

在 C 和 C++ 类型系统中，其行为与 **const** 或 **volatile** 之类的 cv 限定符相同。它产生的特定结果是，无法将类型为 __writeonly 的左值转换为右值。

如果赋值是按读改写的方式实现的，则不允许为 __writeonly 位域赋值。这取决于实现。

示例

```
void foo(__writeonly int *ptr)
{
    *ptr = 0;                          // allowed
    printf("ptr value = %d\n", *ptr); // error
}
```

4.2 __declspec 属性

可以使用 __declspec 关键字指定对象和函数的特殊属性。例如，可以使用 __declspec 关键字声明已导入或导出的函数和变量，或者声明*线程局部存储* (TLS) 对象。

__declspec 关键字必须放在声明规范前面。例如：

```
__declspec(noreturn) void overflow(void);
__declspec(thread) int i;
```

表 4-2 简要说明了可用 __declspec 属性。__declspec 属性是存储类修饰符。它们不影响函数或变量的类型。

表 4-2 编译器支持的 __declspec 属性及其等效项

__declspec 属性	非 __declspec 等效项
__declspec(dllexport)	-
__declspec(dllimport)	-
__declspec(noinline)	__attribute__((noinline))
__declspec(noreturn)	__attribute__((noreturn))
__declspec(nothrow)	-
__declspec(notshared)	-
__declspec(thread)	-

4.2.1 __declspec(dllexport)

构建 DLL 库时，__declspec(dllexport) 属性通过动态符号表导出符号定义。在类上，它用于控制类障碍的可见性，如 vtable、构造 vtable 和 RTTI，并设置成员函数和静态数据成员的缺省可见性。

用法

您可在函数、类或单个类成员上使用 __declspec(dllexport)。

如果将某个内联函数标记为 __declspec(dllexport)，则可能内联该函数定义，但始终使用与非内联函数相同的方法生成和导出该函数的外联实例。

如果将某个类标记为 __declspec(dllexport)（如 class __declspec(dllexport) S { ... };），则将导出该类的全部静态数据成员和成员函数。如果将单个静态数据成员和成员函数都标记为 __declspec(dllexport)，则只导出这些成员。 vtable、构造 vtable 表和 RTTI 也将被导出。

注意

以下声明是正确的：

```
class __declspec(dllexport) S { ... };
```

以下声明是错误的：

```
__declspec(dllexport) class S { ... };
```

结合使用 --export_all_vtbl 和 __declspec(notshared)，可避免导出某一个类或结构的 vtable、构造 vtable 表和 RTTI。 --export_all_vtbl 和 __declspec(dllexport) 一般不同时使用。

限制

如果将一个类标记为 __declspec(dllexport)，则不能将该类的单个成员标记为 __declspec(dllexport)。

如果将一个类标记为 __declspec(dllexport)，则确保将该类的所有基类都标记为 __declspec(dllexport)。

如果要导出一个类中的虚拟函数，则确保导出该类的所有虚拟函数，或者将它们定义为内联以对客户端可见。

示例

声明中所需的 __declspec() 取决于该定义是否位于同一共享库中。

```
/* This is the declaration for use in the same shared library as the */
/* definition */
__declspec(dllexport) extern int mymod_get_version(void);

/* Translation unit containing the definition */
__declspec(dllexport) extern int mymod_get_version(void)
{
   return 42;
}
```

```
/* This is the declaration for use in a shared library that does not contain */
/* the definition */
__declspec(dllimport) extern int mymod_get_version(void);
```

运行下列宏后，定义链接单元中的非定义转换单元将会看到 `__declspec(dllexport)`。

```
/* mymod.h - interface to my module */
#ifdef BUILDING_MYMOD
#define MYMOD_API __declspec(dllexport)
#else /* not BUILDING_MYMOD */
#define MYMOD_API __declspec(dllimport)
#endif

MYMOD_API int mymod_get_version(void);
```

另请参阅

- *__declspec(dllimport)*
- 第4-29 页的*__declspec(notshared)*
- 第2-54 页的*--export_all_vtbl*, *--no_export_all_vtbl*

4.2.2 __declspec(dllimport)

生成 DLL 库时， `__declspec(dllimport)` 属性通过动态符号表导入符号。

用法

如果将内联函数标记为 `__declspec(dllimport)`，则可能内联此编译单元中的函数定义，但不会外联生成该定义。外联调用或地址引用使用导入的符号。

您只能在 **extern** 函数和变量以及类上使用 `__declspec(dllimport)`。

如果将某个类标记为 `__declspec(dllimport)`，则将导入该类的所有静态数据成员和成员函数。如果将单个静态数据成员和成员函数标记为 `__declspec(dllimport)`，则只导入这些成员。

限制

如果将一个类标记为 `__declspec(dllimport)`，则不能将该类的单个成员标记为 `__declspec(dllimport)`。

示例

```
__declspec(dllimport) int i;

class __declspec(dllimport) X
{
  void f();
};
```

另请参阅

- 第4-24 页的*__declspec(dllexport)*

4.2.3 __declspec(noinline)

__declspec(noinline) 属性禁止在函数调用点处内联函数。

注意

此 __declspec 属性具有等效的函数属性 __attribute__((noinline))。

另请参阅

- 第4-63 页的*#pragma inline*, *#pragma no_inline*
- 第4-35 页的*__attribute__((noinline))*

4.2.4 __declspec(noreturn)

__declspec(noreturn) 属性指明函数从不返回值。

注意

此属性具有等效的函数 __attribute((noreturn))。但是， __attribute((noreturn)) 和 __declspec(noreturn) 的不同之处在于，在编译函数定义时，如果函数到达显式或隐式返回，则 __attribute((noreturn)) 将被忽略，同时编译器生成警告消息。而 __declspec(noreturn) 不会被忽略。

用法

可以使用此属性减少调用从不返回值的函数 （如 exit()）所产生的开销。如果 noreturn 函数将值返回到其调用方，则该行为是未定义的。

限制

在调用 noreturn 函数时，不会保留返回地址。这会限制调试器显示调用堆栈的功能。

示例

```
__declspec(noreturn) void overflow(void); // never return on overflow
int negate(int x)
{
    if (x == 0x80000000) overflow();
    return -x;
}
```

另请参阅

- 第4-36 页的*__attribute__((noreturn))*

4.2.5 __declspec(nothrow)

__declspec(nothrow) 属性指明函数调用从不导致将 C++ 异常从调用传播到调用方。

ARM 库头自动将此限定符添加到 C 函数声明中，依照 ISO C 标准，这些函数从不抛出异常。

用法

如果编译器知道函数从不抛出异常，它或许可以为该函数的调用方生成较小的异常处理表。

限制

如果函数调用导致将 C++ 异常从调用传播到调用方，则该行为是未定义的。

如果在不启用异常的情况下进行编译，则会忽略此修饰符。

示例

```
struct S
{
    ~S();
};
__declspec(nothrow) extern void f(void);
void g(void)
```

```
{
    S s;
    f();
}
```

另请参阅

- 第2-57 页的*--force_new_nothrow, --no_force_new_nothrow*
- 第5-13 页的*使用 ::operator new 函数*

4.2.6 __declspec(notshared)

__declspec(notshared) 属性可防止从特定类导出虚拟函数表和 RTTI。这将一直保持为真，而与应用的其他选项无关。例如，使用 --export_all_vtbl 不会覆盖 __declspec(notshared)。

示例

```
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)

__declspec(thread) 属性指明变量是线程局部变量并具有*线程存储时限*，以便链接器安排在创建线程时自动分配的存储。

注意

支持将 __thread 关键字作为 __declspec(thread) 的同义词。

限制

无法动态初始化文件范围的线程局部变量。

示例

```
__declspec(thread) int i;
__thread int j;           // same as __decspec(thread) int j;
```

4.3 函数属性

可以使用 __attribute__ 关键字指定变量或结构字段、函数和类型的特殊属性。此关键字的格式为：

```
__attribute__ ((attribute1, attribute2, ...))
__attribute__ ((__attribute1__, __attribute2__, ...))
```

例如：

```
void * Function_Attributes_malloc_0(int b) __attribute__ ((malloc));
static int b __attribute__ ((__unused__));
```

表 4-3 简要说明了可用函数属性。

表 4-3 编译器支持的函数属性及其等效项

函数属性	非属性等效项
__attribute__((alias))	-
__attribute__((always_inline))	__forceinline
__attribute__((const))	__pure
__attribute__((deprecated))	-
__attribute__((malloc))	-
__attribute__((noinline))	__declspec(noinline)
__attribute__((no_instrument_function))	
__attribute__((nomerge))	-
__attribute__((nonnull))	
__attribute__((noreturn))	__declspec(noreturn))
__attribute__((notailcall))	-
__attribute__((pure))	-
__attribute__((unused))	-
__attribute__((used))	-
__attribute__((visibility("*visibility_type*")))	
__attribute__((weak))	__weak

4.3.1 __attribute__((alias))

可以使用此函数属性为函数指定多个别名。

如果函数是在当前转换单元中定义的，则会将别名调用替换为函数调用，并将别名与原始名称一起发出。如果函数不是在当前转换单元中定义的，则会将别名调用替换为对真实函数的调用。如果将某个函数定义为 **static**，则会将函数名称替换为别名；如果将别名声明为外部别名，则会将该函数声明为外部函数。

注意

此函数属性是 ARM 编译器支持的 GNU 编译器扩展。

注意

也可以使用相应变量属性 __attribute__((alias)) 为变量名称指定别名。

语法

```
return-type newname([argument-list]) __attribute__((alias("oldname")));
```

其中：

oldname 是要指定别名的函数的名称

newname 是已指定别名的函数的新名称。

示例

```
#include <stdio.h>
void foo(void)
{
    printf("%s\n", __FUNCTION__);
}
void bar(void) __attribute__((alias("foo")));
void gazonk(void)
{
    bar(); // calls foo
}
```

另请参阅

- 第4-47 页的*__attribute__((alias))*

4.3.2 __attribute__((always_inline))

此函数属性指示必须内联函数。

编译器将尝试内联函数，而不考虑函数的特性。但是，如果这样做导致出现问题，编译器将不内联函数。例如，递归函数仅内联到本身一次。

注意

此函数属性是 ARM 编译器支持的 GNU 编译器扩展。它具有等效的关键字 __forceinline。

示例

```
static int max(int x, int y) __attribute__((always_inline))
{
    return x > y ? x : y; // always inline if possible
}
```

另请参阅

- 第2-58 页的*--forceinline*
- 第4-6 页的*__forceinline*

4.3.3 __attribute__((const))

很多函数只检查传递给它们的参数，并且只影响返回值。这是一个比 __attribute__((pure)) 严格得多的类，因为不允许函数读取全局内存。如果已知函数只能依靠其自变量起作用，则可以对其进行公共子表达式删除和循环优化。

注意

此函数属性是 ARM 编译器支持的 GNU 编译器扩展。它具有等效的关键字 __pure。

示例

```
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;
}
```

在此代码中，只能调用一次 Function_Attributes_const_0，并将结果加倍以获得正确的返回值。

另请参阅

- 第4-37 页的*__attribute__((pure))*
- *《编译器用户指南》*中第5-13 页的*__pure*。

4.3.4 __attribute__((deprecated))

此函数属性指示存在某个函数，但如果使用这个不提倡使用的函数，编译器必须生成警告。

——注意——

此函数属性是 ARM 编译器支持的 GNU 编译器扩展。

示例

```
int Function_Attributes_deprecated_0(int b) __attribute__ ((deprecated));
```

4.3.5 __attribute__((malloc))

此函数属性指示可以像 malloc 一样处理函数，并且可执行关联的优化。

——注意——

此函数属性是 ARM 编译器支持的 GNU 编译器扩展。

示例

```
void * Function_Attributes_malloc_0(int b) __attribute__ ((malloc));
```

4.3.6 __attribute__((noinline))

此函数属性禁止在函数调用点处内联函数。

注意

此函数属性是 ARM 编译器支持的 GNU 编译器扩展。它具有等效的 __declspec __declspec(noinline)。

示例

```
int fn(void) __attribute__((noinline));

int fn(void)
{
    return 42;
}
```

另请参阅

- 第4-63 页的*#pragma inline*, *#pragma no_inline*
- 第4-27 页的*__declspec(noinline)*

4.3.7 __attribute__((no_instrument_function))

此函数属性将指示在使用 --gnu_instrument 进行检测时不包括该函数。

另请参阅

- 第2-67 页的*--gnu_instrument*, *--no_gnu_instrument*

4.3.8 __attribute__((nomerge))

此函数属性防止源代码中对该函数的不同调用在对象代码中合并。

另请参阅

- 第4-37 页的*__attribute__((notailcall))*
- 第2-109 页的*--retain=option*

4.3.9 __attribute__((nonnull))

此函数属性指定不假定为空指针的函数参数。这将使编译器在遇到此类参数时生成警告。

注意

此函数属性是 ARM 编译器支持的 GNU 编译器扩展。

语法

```
__attribute__((nonnull(arg-index, ...)))
```

其中，*arg-index, ...* 表示参数索引列表。

如果未指定参数索引列表，则所有指针参数都被标记为非零。

示例

以下声明是等效的：

```
void * my_memcpy (void *dest, const void *src, size_t len)
__attribute__((nonnull (1, 2)));

void * my_memcpy (void *dest, const void *src, size_t len)
__attribute__((nonnull));
```

4.3.10 __attribute__((noreturn))

此函数属性指示编译器函数不返回值。随后，编译器可通过删除从不会到达的代码来执行优化。

注意

此函数属性是 ARM 编译器支持的 GNU 编译器扩展。它具有等效的 __declspec __declspec(noreturn)。但是，__attribute((noreturn)) 和 __declspec(noreturn) 的不同之处在于，在编译函数定义时，如果函数到达显式或隐式返回，则 __attribute((noreturn)) 将被忽略，同时编译器生成警告消息，而 __declspec(noreturn) 不会被忽略。

示例

```
int Function_Attributes_NoReturn_0(void) __attribute__ ((noreturn));
```

另请参阅

- 第4-27 页的*__declspec(noreturn)*

4.3.11 __attribute__((notailcall))

此函数属性防止函数的尾调用。也就是说，即使可由普通跳转传送 （因为调用发生在函数的结尾），也始终通过跳转并链接的方式调用函数。

另请参阅

- 第4-35 页的*__attribute__((nomerge))*
- 第2-109 页的*--retain=option*

4.3.12 __attribute__((pure))

很多函数只影响返回值，并且返回值仅取决于参数和全局变量。可以对此类函数进行数据流分析，并且可以将其删除。

注意

此函数属性是 ARM 编译器支持的 GNU 编译器扩展。

虽然此函数属性与 __pure 关键字有关，但它们并*不*等效。__pure 的等效函数属性是 __attribute__((const))。

示例

```
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;
}
```

在此示例中，由于不使用 Function_Attributes_pure_0 调用的结果，因此可以删除该调用。

4.3.13 __attribute__((section("*name*")))

可以使用 section 函数属性将代码放在映像的不同节中。

注意

此函数属性是 ARM 编译器支持的 GNU 编译器扩展。

示例

在以下示例中，将 Function_Attributes_section_0 放在 RO 节 new_section 中，而不是放在 .text 中。

```
void Function_Attributes_section_0 (void)
    __attribute__ ((section ("new_section")));
void Function_Attributes_section_0 (void)
{
    static int aStatic =0;
    aStatic++;
}
```

在以下示例中， section 函数属性覆盖 #pragma arm section 设置。

```
#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
```

另请参阅

- 第4-56 页的*#pragma arm section [section_sort_list]*

4.3.14 __attribute__((unused))

unused 函数属性禁止编译器在未引用该函数时生成警告。这不会更改删除未使用函数的过程的行为。

注意

此函数属性是 ARM 编译器支持的 GNU 编译器扩展。

示例

```
static int Function_Attributes_unused_0(int b) __attribute__ ((unused));
```

4.3.15 __attribute__((used))

此函数属性指示编译器在对象文件中保留静态函数，即使将该函数解除引用也是如此。

标记为已使用的静态函数将按照其声明顺序发出到单个节。可以使用 __attribute__((section)) 指定将函数放置到的节。

注意

此函数属性是 ARM 编译器支持的 GNU 编译器扩展。

注意

也可以使用 __attribute__((used)) 将静态变量标记为已使用。

示例

```
static int lose_this(int);
static int keep_this(int) __attribute__((used));     // retained in object file
static int keep_this_too(int) __attribute__((used)); // retained in object file
```

另请参阅

- 第4-38 页的*__attribute__((section("name")))*
- 第4-52 页的*__attribute__((used))*

4.3.16 __attribute__((visibility("*visibility_type*")))

此函数属性影响 ELF 符号的可见性。

注意

此属性是 ARM 编译器支持的 GNU 编译器扩展。

语法

```
__attribute__((visibility("visibility_type")))
```

其中，*visibility_type* 是下列值之一：

`default` 假定的符号可见性可通过其他选项进行更改。缺省可见性将覆盖此类更改。缺省可见性与外部链接对应。

`hidden` 该符号不存放在动态符号表中，因此，其他可执行文件或共享库都无法直接引用它。使用函数指针可进行间接引用。

`internal` 除非由 *特定于处理器的应用二进制接口* (psABI) 指定，否则，内部可见性意味着不允许从另一模块调用该函数。

`protected` 该符号存放在动态符号表中，但定义模块内的引用将与局部符号绑定。也就是说，另一模块无法覆盖该符号。

用法

除指定 `default` 可见性外，此属性都可与在这些情况下具有外部链接的声明结合使用。

您可在 C 和 C++ 中使用此属性。在 C++ 中，还可将它应用于类型、成员函数和命名空间声明。

示例

```
void __attribute__((visibility( "internal" ))) foo()
{
   ...
}
```

另请参阅

- 第2-8 页的*--arm_linux*
- 第4-53 页的 *__attribute__((visibility("visibility_type")))*

4.3.17 __attribute__((weak))

使用 __attribute__((weak)) 定义的函数将弱导出其符号。

如果使用 __attribute__((weak)) 声明函数，但随后没有使用 __attribute__((weak)) 对其进行定义，则此函数与*弱*函数的行为相同。它与 __weak 关键字的行为不同。

注意

此函数属性是 ARM 编译器支持的 GNU 编译器扩展。

示例

```
extern int Function_Attributes_weak_0 (int b) __attribute__ ((weak));
```

另请参阅

- 第4-20 页的*__weak*

4.4 类型属性

可以使用 __attribute__ 关键字指定变量或结构字段、函数和类型的特殊属性。此关键字的格式为：

```
__attribute__ ((attribute1, attribute2, ...))
__attribute__ ((__attribute1__, __attribute2__, ...))
```

例如：

```
void * Function_Attributes_malloc_0(int b) __attribute__ ((malloc));
static int b __attribute__ ((__unused__));
```

表 4-4 简要说明了可用类型属性。

表 4-4 编译器支持的类型属性及其等效项

类型属性	非属性等效项
__attribute__((bitband))	-
__attribute__((aligned))	__align
__attribute__((packed))	__packed[a]
__attribute__((transparent_union))	-

a. 在 GNU 模式下，__packed 限定符不影响类型。

4.4.1 __attribute__((bitband))

__attribute__((bitband)) 是一种类型属性，可允许对内存体系结构的 SRAM 和外设区中的单个位值进行有效的原子访问。使用此属性，可在特定内存区中通过一次内存访问直接设置或清除单个位，而无需使用传统的读取、修改、写入方式。此外，还可以直接读取单个位而不必使用传统的读取然后移位和屏蔽操作。示例 4-6 演示了 __attribute__((bitband)) 的用法。

示例 4-6 使用 __attribute__((bitband))

```
typedef struct {
    int i: 1;
    int j: 2;
    int k: 3;
} BB __attribute__((bitband));

BB bb __attribute__((at(0x20000004)));
```

```
void foo(void)
{
    bb.i = 1;
}
```

对于区分宽度的外设，分别为位处理操作结构的位域的 **char**、 **short** 和 **int** 类型生成存储或加载到别名空间的字节、半字和字。

在示例 4-7 中，将为 bb.i 生成位处理操作访问。

示例 4-7 位域位处理操作访问

```
typedef struct {
    char i: 1;
    int j: 2;
    int k: 3;
} BB __attribute__((bitband));

BB bb __attribute__((at(0x20000004)));

void foo()
{
    bb.i = 1;
}
```

如果您未使用 __attribute__((at())) 将位处理操作变量存放到位处理操作区，则必须使用其他方法对它进行重新定位。您可使用相应的分散加载描述文件或 --rw_base 链接器命令行选项来进行重定位。有关详细信息，请参阅 《链接器参考指南》。

限制

应用的限制如下：

- 此类型属性仅可与 **struct** 一起使用。任何联合类型或其他带有联合成员的聚合类型都不能进行位处理操作。
- 结构中的成员无法单独进行位处理操作。
- 仅为包含单个位的位域生成位处理操作访问。
- 不对 **const** 对象、指针和局部对象生成位处理操作访问。

另请参阅

- 第4-48 页的*__attribute__((at(address)))*
- *《编译器用户指南》*中第4-15 页的*位处理操作*
- 处理器的 《技术参考手册》 (*Technical Reference Manual*)。

4.4.2 __attribute__((aligned))

aligned 类型属性指定类型的最低对齐要求。

注意

此类型属性是 ARM 编译器支持的 GNU 编译器扩展。

4.4.3 __attribute((packed))

packed 类型属性指定类型必须具有最小的可能对齐要求。

注意

此类型属性是 ARM 编译器支持的 GNU 编译器扩展。

错误

如果在 typedef 中使用此属性，编译器将生成警告消息。

另请参阅

- 第4-11 页的*__packed*
- 第4-65 页的*#pragma pack(n)*
- 第5-10 页的*压缩结构*
- *《编译器用户指南》*中第5-25 页的*__packed 限定符和未对齐的数据访问*
- *《编译器用户指南》*中第5-27 页的*__packed 结构与单个 __packed 字段*。

4.4.4 __attribute__((transparent_union))

可以使用 transparent_union 类型属性指定 *transparent_union* 类型，即使用 __attribute__((transparent_union))__ 限定的联合数据类型。

使用具有透明联合类型的参数定义函数时，如果函数调用使用的自变量具有联合中的任何类型，则会导致初始化具有以下成员的联合对象：成员具有传递的自变量类型，并将其值设置为传递的自变量值。

使用 __attribute__((transparent_union)) 限定联合数据类型时，透明联合将应用于所有具有该类型的函数参数。

注意

此类型属性是 ARM 编译器支持的 GNU 编译器扩展。

注意

也可以使用相应 __attribute__((transparent_union)) 变量属性限定各个函数参数。

示例

```
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 */
}
```

模式

仅在 GNU 模式下支持。

另请参阅

- 第4-51 页的*__attribute__((transparent_union))*

4.5 变量属性

可以使用 __attribute__ 关键字指定变量或结构字段、函数和类型的特殊属性。此关键字的格式为：

```
__attribute__ ((attribute1, attribute2, ...))
__attribute__ ((__attribute1__, __attribute2__, ...))
```

例如：

```
void * Function_Attributes_malloc_0(int b) __attribute__ ((malloc));
static int b __attribute__ ((__unused__));
```

第4-31 页的表 4-3 简要说明了可用变量属性。

表 4-5 编译器支持的变量属性及其等效项

变量属性	非属性等效项
__attribute__((alias))	-
__attribute__((at(*address*)))	-
__attribute__((aligned))	-
__attribute__((deprecated))	-
__attribute__((packed))	-
__attribute__((section(“name”)))	-
__attribute__((transparent_union))	-
__attribute__((unused))	-
__attribute__((used))	-
__attribute__((weak))	__weak
__attribute__((visibility("*visibility_type*")))	
__attribute__((zeroinit))	-

4.5.1 __attribute__((alias))

可以使用此变量属性为变量指定多个别名。

如果变量是在当前转换单元中定义的，则会将别名引用替换为变量引用，并将别名与原始名称一起发出。如果变量不是在当前转换单元中定义的，则会将别名引用替换为对真实变量的引用。如果将某个变量定义为 **static**，则会将变量名称替换为别名；如果将别名声明为外部别名，则会将该变量声明为外部变量。

注意

也可以使用相应函数属性 __attribute__((alias)) 为函数名称指定别名。

语法

```
type newname __attribute__((alias("oldname")));
```

其中：

oldname 是要指定别名的变量的名称

newname 是已指定别名的变量的新名称。

示例

```
#include <stdio.h>
int oldname = 1;
extern int newname __attribute__((alias("oldname"))); // declaration
void foo(void)
{
    printf("newname = %d\n", newname); // prints 1
}
```

另请参阅

- 第4-32 页的*__attribute__((alias))*

4.5.2 __attribute__((at(*address*)))

可以使用此变量属性指定变量的绝对地址。

变量放在其自己的节中，编译器将为包含变量的节指定适当的类型：

- 只读变量放在 RO 类型的节中。
- 已初始化的读写变量放在 RW 类型的节中。

  特别地，显式初始化为零的变量放在 RW 中，而不是放在 ZI 中。此类变量不适合编译器的 ZI 到 RW 优化。
- 未初始化的变量放在 ZI 类型的节中。

注意

GNU 编译器不支持此变量属性。

语法

```
__attribute__((at(address)))
```

其中：

address 是所需的变量地址。

限制

链接器并非始终能够放置 at 变量属性生成的节。

错误

如果无法将节放置在指定地址，链接器将显示一条错误消息。

示例

```
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 */
```

另请参阅

- *《链接器用户指南》*中第5-16 页的*使用 __at 节将节放在特定地址中*。

4.5.3 __attribute__((aligned))

aligned 变量属性指定变量或结构字段的最低对齐要求 （按字节计算）。

注意

此变量属性是 ARM 编译器支持的 GNU 编译器扩展。

示例

```
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 */
```

另请参阅

- 第4-2 页的*__align*

4.5.4 __attribute__((deprecated))

可以使用 deprecated 变量属性声明不提倡使用的变量，而不会导致编译器发出任何警告或错误。但是，对 deprecated 变量的任何访问都会生成警告，但仍会进行编译。警告指出了使用和定义变量的位置。这有助于确定不提倡使用特定定义的原因。

注意

此变量属性是 ARM 编译器支持的 GNU 编译器扩展。

示例

```
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;
}
```

编译此示例时，将生成两条警告消息。

4.5.5 __attribute__((packed))

packed 变量属性指定变量或结构字段具有最小的可能对齐要求。即，除非使用 aligned 属性指定更大的值，否则，变量为一个字节，字段为一位。

注意

此变量属性是 ARM 编译器支持的 GNU 编译器扩展。

示例

```
struct
{
    char a;
    int b __attribute__ ((packed));
} Variable_Attributes_packed_0;
```

另请参阅

- 第4-65 页的*#pragma pack(n)*
- 第5-10 页的*压缩结构*
- *《编译器用户指南》*中第5-25 页的*__packed 限定符和未对齐的数据访问*
- *《编译器用户指南》*中第5-27 页的*__packed 结构与单个 __packed 字段*。

4.5.6 __attribute__((section("name")))

通常， ARM 编译器将它生成的对象放在节中，如 data 和 bss。但是，您可能需要使用其他数据节，或者希望变量出现在特殊节中，例如，便于映射到特殊硬件。section 属性指定变量必须放在特定数据节中。如果使用 section 属性，则将只读变量放在 RO 数据节中，而将读写变量放在 RW 数据节中，除非您使用 zero_init 属性。在这种情况下，变量被放在 ZI 节中。

注意

此变量属性是 ARM 编译器支持的 GNU 编译器扩展。

示例

```
/* in RO 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))

如果将 transparent_union 变量属性附加到具有联合类型的函数参数中，则表示相应自变量可具有任何联合成员的类型，但传递自变量时就好像其类型是第一个联合成员的类型一样。

注意

C 规范规定，如果将联合作为一种类型写入并以另一种类型回读，则返回的值是未定义的。因此，区分 transparent_union 以哪种类型写入的方法也必须作为自变量传递。

注意

此变量属性是 ARM 编译器支持的 GNU 编译器扩展。

注意

也可以在 typedef 上将此属性用于联合数据类型。在这种情况下，它应用于所有该类型的函数参数。

模式

仅在 GNU 模式下支持。

示例

```
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);
}
```

另请参阅

- 第4-44 页的*__attribute__((transparent_union))*

4.5.8 __attribute__((unused))

通常，如果声明了某个变量，但从未对其进行引用，编译器将发出警告。此属性指示编译器您预计不会使用某个变量，并指示它在未使用该变量时不要发出警告。

注意

此变量属性是 ARM 编译器支持的 GNU 编译器扩展。

示例

```
void Variable_Attributes_unused_0()
{
    static int aStatic =0;
    int aUnused __attribute__ ((unused));
    int bUnused;
    aStatic++;
}
```

在此示例中，编译器发出已声明但从未引用 `bUnused` 的警告，但不会发出有关 `aUnused` 的警告。

注意

GNU 编译器不会生成任何警告。

4.5.9 __attribute__((used))

此变量属性指示编译器在对象文件中保留某个静态变量，即使解除了对该变量的引用也是如此。

标记为已使用的静态变量将按照其声明顺序发出到单个节。可以使用 `__attribute__((section))` 指定将变量放置到的节。

——注意——
此变量属性是 ARM 编译器支持的 GNU 编译器扩展。

——注意——
也可以使用 __attribute__((used)) 将静态函数标记为已使用。

用法

可以使用 __attribute__((used)) 在对象中生成表。

示例

```
static int lose_this = 1;
static int keep_this __attribute__((used)) = 2;     // retained in object file
static int keep_this_too __attribute__((used)) = 3; // retained in object file
```

另请参阅

- 第4-50 页的*__attribute__((section("name")))*
- 第4-39 页的*__attribute__((used))*

4.5.10 __attribute__((visibility("*visibility_type*")))

此变量属性影响 ELF 符号的可见性。

——注意——
此属性是 ARM 编译器支持的 GNU 编译器扩展。

visibility_type 的可能值与为同名的函数属性指定的值相同。

示例

```
int i __attribute__((visibility( “hidden” )));
```

另请参阅

- 第2-8 页的*--arm_linux*
- 第4-40 页的 *__attribute__((visibility("visibility_type")))*

4.5.11 __attribute__((weak))

允许声明弱变量，使用的方法与 __weak 类似。

- 在 GNU 模式下：

```
extern int Variable_Attributes_weak_1 __attribute__((weak));
```

- 非 GNU 模式下的等效项为：

```
__weak int Variable_Attributes_weak_compare;
```

注意

在 GNU 模式下，需要使用 extern 限定符。在非 GNU 模式下，编译器假定，如果变量不是 extern，则像任何其他非弱变量一样进行处理。

注意

此变量属性是 ARM 编译器支持的 GNU 编译器扩展。

另请参阅

- 第4-20 页的*__weak*

4.5.12 __attribute__((zero_init))

section 属性指定变量必须放在特定数据节中。zero_init 属性指定将没有初始值设定项的变量放在 ZI 数据节中。如果指定了初始值设定项，则会报告错误。

示例

```
__attribute__((zero_init)) int x;                    /* in section ".bss" */
__attribute__((section("mybss"), zero_init)) int y;  /* in section "mybss" */
```

另请参阅

- 第4-50 页的*__attribute__((section("name")))*

4.6 编译指示

ARM 编译器可识别很多 ARM 特定的编译指示。表 4-6 简要说明了可用编译指示。

——注意——

编译指示优先于相关的命令行选项。例如，`#pragma arm` 覆盖 `--thumb` 命令行选项。

表 4-6 编译器支持的编译指示

编译指示		
`#pragma anon_unions`, `#pragma no_anon_unions`	`#pragma hdrstop`	`#pragma Otime`
`#pragma arm`	`#pragma import` *symbol_name*	`#pragma pack(`*n*`)`
`#pragma arm section` [*section_sort_list*]	`#pragma import(__use_full_stdio)`	`#pragma pop`
`#pragma diag_default` *tag*[,*tag*,...]	`#pragma inline`, `#pragma no_inline`	`#pragma push`
`#pragma diag_error` *tag*[,*tag*,...]	`#pragma no_pch`	`#pragma softfp_linkage`, `no_softfp_linkage`
`#pragma diag_remark` *tag*[,*tag*,...]	`#pragma O`*num*	`#pragma unroll [(`*n*`)]`
`#pragma diag_suppress` *tag*[,*tag*,...]	`#pragma once`	`#pragma unroll_completely`
`#pragma diag_warning` *tag*[,*tag*,...]	`#pragma hdrstop`	`#pragma thumb`
`#pragma` [no_]exceptions_unwind	`#pragma Ospace`	

4.6.1 #pragma anon_unions, #pragma no_anon_unions

这些编译指示启用和禁用对匿名结构和联合的支持。

缺省设置

缺省值为 #pragma no_anon_unions。

另请参阅

- 第3-19 页的*匿名类、结构和联合*
- 第4-44 页的*__attribute__((transparent_union))*

4.6.2 #pragma arm

此编译指示将代码生成切换为 ARM 指令集。它覆盖 --thumb 编译器选项。

另请参阅

- 第2-8 页的*--arm*
- 第2-119 页的*--thumb*
- 第4-70 页的*#pragma thumb*

4.6.3 #pragma arm section [*section_sort_list*]

此编译指示指定要用于后续函数或对象的节名称。这包括编译器为进行初始化而创建的匿名对象的定义。

注意

可以将 __attribute__((section(..))) 用于函数或变量以替代 #pragma arm section。

语法

#pragma arm section [*section_sort_list*]

其中：

section_sort_list 指定要用于后续函数或对象的节名称的可选列表。*section_sort_list* 的语法为：

section_type[[=]"*name*"] [,*section_type*="*name*"]*

有效的节类型是：

- code
- rodata
- rwdata
- zidata。

用法

可以将分散加载描述文件与 ARM 链接器配合使用，以控制将已命名的节放在特定内存地址的方式。

限制

此选项对以下内容无效：

- 内联函数及其局部静态变量。
- 模板实例化及其局部静态变量。
- 删除未使用的变量和函数。但是，可通过使用 #pragma arm section，使链接器能够删除本来可能会保留的函数或变量，因为它与使用的函数或变量位于相同的节中。
- 将定义写入对象文件的顺序。

示例

```
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 rwdata = "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
```

另请参阅

- 第4-38 页的*__attribute__((section("name")))*
- *《链接器用户指南》*中第 5 章 *使用分散加载描述文件*。

4.6.4 #pragma diag_default *tag[,tag,...]*

此编译指示将具有指定标签的诊断消息的严重性恢复为在发出任何编译指示之前有效的严重性。

语法

```
#pragma diag_default tag[,tag,...]
```

其中：

tag[,*tag*,...] 是一个以逗号分隔的诊断消息编号列表，用于指定要更改其严重性的消息。

必须至少指定一个诊断消息编号。

示例

```
// <stdio.h> not #included deliberately
#pragma diag_error 223
void hello(void)
{
    printf("Hello ");
}
#pragma diag_default 223
void world(void)
{
    printf("world!\n");
}
```

使用 --diag_warning=223 选项编译此代码时，将生成一些诊断消息以报告 printf() 函数是隐式声明的。

#pragma diag_default 223 的作用是将诊断消息 223 的严重性恢复为警告严重性（由 --diag_warning 命令行选项指定）。

另请参阅

- 第2-48 页的*--diag_warning=tag[,tag,...]*
- 第4-59 页的*#pragma diag_error tag[,tag,...]*
- 第4-59 页的*#pragma diag_remark tag[,tag,...]*

- 第4-60 页的*#pragma diag_suppress tag[,tag,...]*
- 第4-61 页的*#pragma diag_warning tag[, tag, ...]*
- *《编译器用户指南》*中第6-4 页的*控制诊断消息的输出*。

4.6.5 #pragma diag_error *tag[,tag,...]*

此编译指示将具有指定标签的诊断消息设置为错误严重性。

语法

```
#pragma diag_error tag[,tag,...]
```

其中：

tag[,*tag*,...] 是一个以逗号分隔的诊断消息编号列表，用于指定要更改其严重性的消息。

必须至少指定一个诊断消息编号。

另请参阅

- 第2-44 页的*--diag_error=tag[,tag,...]*
- 第4-58 页的*#pragma diag_default tag[,tag,...]*
- *#pragma diag_remark tag[,tag,...]*
- 第4-60 页的*#pragma diag_suppress tag[,tag,...]*
- 第4-61 页的*#pragma diag_warning tag[, tag, ...]*
- *《编译器用户指南》*中第6-5 页的*更改诊断消息的严重性*。

4.6.6 #pragma diag_remark *tag[,tag,...]*

此编译指示将具有指定标签的诊断消息设置为备注严重性。

#pragma diag_remark 的行为与 #pragma diag_errors 类似，只不过编译器将具有指定标签的诊断消息设置为备注严重性，而不是设置为错误严重性。

注意

缺省情况下不显示备注。可以使用 --remarks 编译器选项来查看备注消息。

语法

```
#pragma diag_remark tag[,tag,...]
```

其中：

tag[,*tag*,...]　　是一个以逗号分隔的诊断消息编号列表，用于指定要更改其严重性的消息。

另请参阅

- 第2-45 页的*--diag_remark=tag[,tag,...]*
- 第2-108 页的*--remarks*
- 第4-58 页的*#pragma diag_default tag[,tag,...]*
- 第4-59 页的*#pragma diag_error tag[,tag,...]*
- *#pragma diag_suppress tag[,tag,...]*
- 第4-61 页的*#pragma diag_warning tag[, tag, ...]*
- *《编译器用户指南》*中第6-5 页的*更改诊断消息的严重性*。

4.6.7 #pragma diag_suppress *tag[,tag,...]*

此编译指示禁用所有具有指定标签的诊断消息。

`#pragma diag_suppress` 的行为与 `#pragma diag_errors` 类似，只不过编译器禁止具有指定标签的诊断消息，而不是将其设置为具有错误严重性。

语法

`#pragma diag_suppress` *tag*[,*tag*,...]

其中：

tag[,*tag*,...]　　是一个以逗号分隔的诊断消息编号列表，用于指定要禁止的消息。

另请参阅

- 第2-46 页的*--diag_suppress=tag[,tag,...]*
- 第4-58 页的*#pragma diag_default tag[,tag,...]*
- 第4-59 页的*#pragma diag_error tag[,tag,...]*
- 第4-59 页的*#pragma diag_remark tag[,tag,...]*
- 第4-61 页的*#pragma diag_warning tag[, tag, ...]*
- *《编译器用户指南》*中第6-6 页的*禁止显示诊断消息*。

4.6.8 #pragma diag_warning *tag[, tag, ...]*

此编译指示将具有指定标签的诊断消息设置为警告严重性。

#pragma diag_remark 的行为与 #pragma diag_errors 类似，只不过编译器将具有指定标签的诊断消息设置为备注严重性，而不是设置为错误严重性。

语法

#pragma diag_warning *tag*[,*tag*,...]

其中：

tag[,*tag*,...] 是一个以逗号分隔的诊断消息编号列表，用于指定要更改其严重性的消息。

另请参阅

- 第2-48 页的*--diag_warning=tag[,tag,...]*
- 第4-58 页的*#pragma diag_default tag[,tag,...]*
- 第4-59 页的*#pragma diag_error tag[,tag,...]*
- 第4-59 页的*#pragma diag_remark tag[,tag,...]*
- 第4-60 页的*#pragma diag_suppress tag[,tag,...]*
- *《编译器用户指南》*中第6-5 页的*更改诊断消息的严重性*。

4.6.9 #pragma exceptions_unwind, #pragma no_exceptions_unwind

此编译指示在运行时启用和禁用函数展开。

缺省设置

缺省值为 #pragma exceptions_unwind。

另请参阅

- 第2-53 页的*--exceptions*, *--no_exceptions*
- 第2-54 页的*--exceptions_unwind*, *--no_exceptions_unwind*
- 第5-17 页的*运行时的函数展开*

4.6.10 #pragma hdrstop

可以使用此编译指示指定预编译头文件集的结束位置。

此编译指示必须位于第一个不属于预处理指令的标记前面。

另请参阅

- *《编译器用户指南》*中第2-18 页的*预编译的头文件*。

4.6.11 #pragma import *symbol_name*

此编译指示生成对 *symbol_name* 的导入引用。它与以下汇编器指令相同：

```
IMPORT symbol_name
```

语法

```
#pragma import symbol_name
```

其中：

symbol_name 是一个要导入的符号。

用法

可以使用此编译指示选择 C 库的某些功能，如堆实现或实时除法。如果本手册中介绍的功能要求导入符号引用，则会指定所需的符号。

另请参阅

- *《库和浮点支持指南》*中第2-18 页的*使用 C 库构建应用程序*。

4.6.12 #pragma import(__use_full_stdio)

此编译指示选择一个 microlib 的扩展版本，该版本使用完整的标准 ANSI C 输入和输出功能。

将产生以下异常：

- `feof()` 和 `ferror()` 始终返回 `0`
- `setvbuf()` 和 `setbuf()` 必定失败。

`feof()` 和 `ferror()` 始终返回 `0` 是因为错误指示符和文件末尾指示符不受支持。

`setvbuf()` 和 `setbuf()` 必定失败是因为所有流都没有缓冲。

microlib stdio 的这个版本可使用与 standardlib stdio 函数相同的方法重定向目标。

另请参阅

- 第2-79 页的*--library_type=lib*
- *《库和浮点支持指南》*中的第 3 章 *C 微型库*
- *《库和浮点支持指南》*中第2-76 页的*调整输入/输出函数*。

4.6.13 #pragma inline, #pragma no_inline

这些编译指示控制内联操作，与 --inline 和 --no_inline 命令行选项类似。按照 #pragma no_inline 定义的函数不会内联到其他函数，并且不会内联其自己的调用。

也可以通过将函数标记为 __declspec(noinline) 或 __attribute__((noinline))，实现禁止内联到其他函数。

缺省设置

缺省值为 #pragma inline。

另请参阅

- 第2-73 页的*--inline*, *--no_inline*
- 第4-27 页的*__declspec(noinline)*
- 第4-35 页的*__attribute__((noinline))*

4.6.14 #pragma no_pch

此编译指示禁止对给定源文件进行 PCH 处理。

另请参阅

- 第2-98 页的*--pch*
- *《编译器用户指南》*中第2-18 页的*预编译的头文件*。

4.6.15 #pragma O*num*

此编译指示更改优化级别。

语法

#pragma O*num*

其中：

num 是新的优化级别。
num 的值为 0、 1、 2 或 3。

另请参阅

- 第2-94 页的*-Onum*

4.6.16 #pragma once

此编译指示允许编译器跳过该头文件的后续包含语句。

#pragma once 是可接受的，以便与其他编译器保持兼容，并且允许使用其他形式的头文件保护编码。但是，最好使用 #ifndef 和 #define 编码，因为这更便于进行移植。

示例

以下示例说明了将 #ifndef 保护放在文件体周围并将保护变量的 #define 放在 #ifndef 后面的情形。

```
#ifndef FILE_H
#define FILE_H
#pragma once          // optional ... body of the header file ...#endif
```

在此示例中，将 #pragma once 标记为可选。这是因为编译器可识别 #ifndef 头文件保护编码并跳过后续包含，即使缺少 #pragma once 也是如此。

4.6.17 #pragma Ospace

此编译指示指示编译器执行优化以减小映像大小，但可能会以延长执行时间为代价。

另请参阅

- *#pragma Otime*
- 第2-96 页的*-Ospace*

4.6.18 #pragma Otime

此编译指示指示编译器执行优化以缩短执行时间，但可能会以增加映像大小为代价。

另请参阅

- *#pragma Ospace*
- 第2-97 页的*-Otime*

4.6.19 #pragma pack(*n*)

此编译指示将结构成员对齐到 n 和这些成员的自然对齐边界中的较小值。压缩对象是通过未对齐访问读取和写入的。

语法

```
#pragma pack(n)
```

其中：

n 是以字节为单位的对齐，有效的对齐值为 1、2、4 和 8。

缺省设置

缺省值为 #pragma pack(8)。

示例

该示例演示 pack(2) 如何将整数变量 b 对齐到 2 字节边界。

```
typedef struct
{
    char a;
    int b;
```

```
} S;

#pragma pack(2)

typedef struct
{
    char a;
    int b;
} SP;

S var = { 0x11, 0x44444444 };
SP pvar = { 0x11, 0x44444444 };
```

图 4-1 是 S 的布局；图 4-2 是 SP 的布局。在图 4-2 中，x 表示单字节填充。

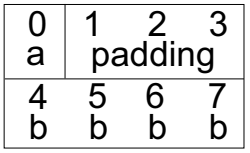

图 4-1 非压缩结构 S

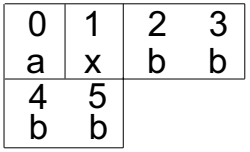

图 4-2 压缩结构 SP

注意

SP 是一个 6 字节结构。b 之后没有填充。

另请参阅

- 第4-11 页的*__packed*
- 第4-50 页的*__attribute__((packed))*
- 第5-10 页的*压缩结构*
- *《编译器用户指南》*中第5-25 页的*__packed 限定符和未对齐的数据访问*
- *《编译器用户指南》*中第5-27 页的*__packed 结构与单个 __packed 字段*。

4.6.20 #pragma pop

此编译指示恢复以前保存的编译指示状态。

另请参阅

- *#pragma push*

4.6.21 #pragma push

此编译指示保存当前编译指示状态。

另请参阅

- *#pragma pop*

4.6.22 #pragma softfp_linkage, #pragma no_softfp_linkage

这些编译指示控制软件浮点链接。

#pragma softfp_linkage 指明到下一个 #pragma no_softfp_linkage 之前的所有函数声明描述了使用软件浮点链接的函数。

注意

此编译指示具有等效的关键字 __softfp。

用法

若要应用于整个接口说明 （位于头文件中）而不改变该文件，此编译指示可能非常有用。

缺省设置

缺省值为 #pragma no_softfp_linkage。

另请参阅

- 第4-15 页的*__softfp*
- *《编译器用户指南》*中第5-35 页的*浮点计算和链接*。

4.6.23 #pragma unroll [(*n*)]

此编译指示指示编译器通过 *n* 次迭代展开循环。

注意

使用 #pragma unroll [(*n*)] 可以展开向量化和非向量化的循环。即， #pragma unroll [(*n*)] 同时适用于 --vectorize 和 --no_vectorize。

语法

```
#pragma unroll
#pragma unroll (n)
```

其中：

n 是一个可选值，用于指示要展开的迭代次数。

缺省设置

如果没有指定 *n* 的值，则编译器假定为 #pragma unroll (4)。

用法

使用 -O3 -Otime 进行编译时，如果编译器认为展开循环比较有利，则会自动将其展开。可以使用此编译指示请求编译器展开未自动展开的循环。

注意

仅当有证据表明 （例如，从 --diag_warning=optimizations 中），编译器本身没有以最优方式展开循环时，才应使用此 #pragma。

限制

只能在 **for** 循环、 **while** 循环或 **do** ... **while** 循环的紧前面使用 #pragma unroll [(*n*)]。

示例

```
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;
        }
    }
}
```

在此示例中，编译器没有正常完成其循环分析，因为 src2 是以 src2[j][k] 进行索引的，而循环是按相反顺序嵌套的，即，j 位于 k 内部。如果在示例中取消 #pragma unroll 注释，编译器将继续展开循环四次。

如果目的是增加大小不是 4 的倍数的矩阵 （例如 *n* * *n* 矩阵），则可以改用 #pragma unroll (*m*)，其中， *m* 是某个值，以使 *n* 是 *m* 的整数倍。

另请参阅

- 第2-48 页的*--diag_warning=optimizations*
- 第2-94 页的*-Onum*
- 第2-97 页的*-Otime*
- 第2-127 页的*--vectorize*, *--no_vectorize*
- *#pragma unroll_completely*
- *《编译器用户指南》*中第5-4 页的*优化循环*。

4.6.24 #pragma unroll_completely

此编译指示指示编译器完全展开循环。仅当编译器可以确定循环包含的迭代次数时，它才有效。

注意

使用 #pragma unroll_completely 可以展开向量化和非向量化循环。即， #pragma unroll_completely 适用于 --no_vectorize 和 --vectorize。

用法

使用 -O3 -Otime 进行编译时，如果编译器认为展开循环比较有利，则会自动将其展开。可以使用此编译指示请求编译器完全展开未自动完全展开的循环。

注意

仅当有证据表明（例如，从 --diag_warning=optimizations 中），编译器本身没有以最优方式展开循环时，才应使用此 #pragma。

限制

只能在 **for** 循环、**while** 循环或 **do** ... **while** 循环的紧前面使用 #pragma unroll_completely。

在外部循环中使用 #pragma unroll_completely 可防止向量化。另一方面，在某些情况下，在内部循环中使用 #pragma unroll_completely 可能会有所帮助。

另请参阅

- 第2-48 页的*--diag_warning=optimizations*
- 第2-94 页的*-Onum*
- 第2-97 页的*-Otime*
- 第2-127 页的*--vectorize*, *--no_vectorize*
- 第4-68 页的*#pragma unroll [(n)]*
- *《编译器用户指南》*中第5-4 页的*优化循环*。

4.6.25 #pragma thumb

此编译指示将代码生成切换为 Thumb 指令集。它覆盖 --arm 编译器选项。

如果要针对 Thumb 2 以前的处理器使用 VFP 编译代码，则会为 ARM 编译所有包含浮点运算的函数。

另请参阅

- 第2-8 页的*--arm*
- 第2-119 页的*--thumb*
- 第4-56 页的*#pragma arm*

4.7 指令内在函数

本节介绍 C 或 C++ 代码中用于实现 ARM 机器语言指令的指令内在函数。表 4-7 简要说明了可用内在函数。

表 4-7 ARM 编译器支持的指令内在函数

指令内在函数		
__breakpoint	__ldrt	__schedule_barrier
__cdp	__memory_changed	__semihost
__clrex	__nop	__sev
__clz	__pld	__sqrt
__current_pc	__pldw	__sqrtf
__current_sp	__pli	__ssat
__disable_fiq	__promise	__strex
__disable_irq	__qadd	__strexd
__enable_fiq	__qdbl	__strt
__enable_irq	__qsub	__swp
__fabs	__rbit	__usat
__fabsf	__rev	__wfe
__force_stores	__return_address	__wfi
__ldrex	__ror	__yield
__ldrexd		

另请参阅第4-112 页的*GNU 内置函数*。

4.7.1 __breakpoint

此内在函数在编译器生成的指令流中插入 BKPT 指令。它允许在 C 或 C++ 代码中包含断点指令。

语法

```
void __breakpoint(int val)
```

其中：

val 是编译时常数整数，其范围是：

- `0 ... 65535` 如果要将源代码编译为 ARM 代码
- `0 ... 255` 如果要将源代码编译为 Thumb 代码。

错误

为不支持 BKPT 指令的目标进行编译时，编译器无法识别 __breakpoint 内在函数。在这种情况下，编译器将生成警告或错误。

如果在不支持 BKPT 指令的体系结构上执行该指令，则会生成未定义的指令陷阱。

示例

```
void func(void)
{
    ...
    __breakpoint(0xF02C);
    ...
}
```

另请参阅

- *《汇编器指南》*中第4-128 页的*BKPT*。

4.7.2 __cdp

此内在函数在编译器生成的指令流中插入 CDP 或 CDP2 指令。它允许在 C 或 C++ 代码中包含协处理器数据运算。

语法

```
__cdp(unsigned int coproc, unsigned int opcode1, unsigned int opcode2)
```

其中：

coproc 确定指令适用的协处理器。

coproc 必须是一个在 0 到 15 范围内的整数。

opcode1 是一个协处理器特定的操作码。
将 0x100 添加到操作码中以生成 CDP2 指令。

opcode2 是一个协处理器特定的操作码。

用法

这些指令的具体用法取决于协处理器。有关详细信息，请参阅协处理器文档。

另请参阅

- *《汇编器指南》*中第4-120 页的*CDP 和 CDP2*。

4.7.3 __clrex

此内在函数在编译器生成的指令流中插入 CLREX 指令。它允许在 C 或 C++ 代码中包含 CLREX 指令。

语法

```
void __clrex(void)
```

错误

为不支持 CLREX 指令的目标进行编译时，编译器无法识别 __clrex 内在函数。在这种情况下，编译器将生成警告或错误。

另请参阅

- *《汇编器指南》*中第4-37 页的*CLREX*。

4.7.4 __clz

此内在函数在编译器生成的指令流中插入 CLZ 指令或等效的代码序列。它允许在 C 或 C++ 代码中计算数据值的前导零个数。

语法

```
unsigned char __clz(unsigned int val)
```

其中：
val 是一个 **unsigned int**。

返回值

__clz 内在函数返回 *val* 中的前导零个数。

另请参阅

- 第4-114 页的*其他内置函数*
- *《汇编器指南》*中第4-52 页的*CLZ*。

4.7.5 __current_pc

通过使用此内在函数，可以确定程序中使用该内在函数的位置处的程序计数器的当前值。

语法

```
unsigned int __current_pc(void)
```

返回值

__current_pc 内在函数返回程序中使用该内在函数的位置处的程序计数器的当前值。

另请参阅

- *__current_sp*
- 第4-89 页的*__return_address*
- *《编译器用户指南》*中第7-25 页的*访问 sp、 lr 或 pc 的旧内联汇编器*

4.7.6 __current_sp

此内在函数返回程序中当前位置的堆栈指针值。

语法

```
unsigned int __current_sp(void)
```

返回值

__current_sp 内在函数返回程序中使用该内在函数的位置处的堆栈指针的当前值。

另请参阅

- 第4-114 页的*其他内置函数*
- 第4-74 页的*__current_pc*
- 第4-89 页的*__return_address*
- *《编译器用户指南》*中第7-25 页的*访问 sp、 lr 或 pc 的旧内联汇编器*

4.7.7 __disable_fiq

此内在函数禁用 FIQ 中断。

注意

通常，此内在函数通过设置 CPSR 中的 F 位禁用 FIQ 中断。但对于 v7-M，它设置故障掩码寄存器 (FAULTMASK)。 v6-M 中不支持 FIQ 中断。

语法

```
int __disable_fiq(void)
```

注意

在 M-profile 上， __disable_fiq 内在函数具有以下原型：

```
void __disable_fiq(void)
```

返回值

__disable_fiq 返回在禁用 FIQ 中断之前 FIQ 中断掩码在 PSR 中包含的值。

限制

只能在特权模式 （即非用户模式）下执行 __disable_fiq 内在函数。在用户模式下，此内在函数不会更改 CPSR 中的中断标记。

示例

```
void foo(void)
{
    int was_masked = __disable_fiq();
    /* ... */
    if (!was_masked)
        __enable_fiq();
}
```

另请参阅

- 第4-77 页的*__enable_fiq*

4.7.8 __disable_irq

此内在函数禁用 IRQ 中断。

注意

通常，此内在函数通过设置 CPSR 中的 I 位禁用 IRQ 中断。但对于 M-profile，它设置异常掩码寄存器 (PRIMASK)。

语法

```
int __disable_irq(void)
```

注意

在 M-profile 上， __disable_irq 内在函数具有以下原型：

```
void __disable_irq(void)
```

返回值

__disable_irq() 返回在禁用 IRQ 中断之前 IRQ 中断掩码在 PSR 中包含的值。

示例

```
void foo(void)
{
    int was_masked = __disable_irq();
    /* ... */
    if (!was_masked)
        __enable_irq();
}
```

限制

只能在特权模式 （即非用户模式）下执行 __disable_irq 内在函数。在用户模式下，此内在函数不会更改 CPSR 中的中断标记。

另请参阅

- 第4-77 页的*__enable_irq*

4.7.9 __enable_fiq

此内在函数启用 FIQ 中断。

——注意——

通常，此内在函数通过清除 CPSR 中的 F 位启用 FIQ 中断。但对于 v7-M，它清除故障掩码寄存器 (FAULTMASK)。 v6-M 中不支持 FIQ 中断。

语法

```
void __enable_fiq(void)
```

限制

只能在特权模式 （即非用户模式）下执行 __enable_fiq 内在函数。在用户模式下，此内在函数不会更改 CPSR 中的中断标记。

另请参阅

- 第4-75 页的*__disable_fiq*

4.7.10 __enable_irq

此内在函数启用 IRQ 中断。

——注意——

通常，此内在函数通过清除 CPSR 中的 I 位启用 IRQ 中断。但对于 Cortex M-profile 处理器，它清除异常掩码寄存器 (PRIMASK)。

语法

```
void __enable_irq(void)
```

限制

只能在特权模式 （即非用户模式）下执行 __enable_irq 内在函数。在用户模式下，此内在函数不会更改 CPSR 中的中断标记。

另请参阅

- 第4-76 页的*__disable_irq*

4.7.11 __fabs

此内在函数在编译器生成的指令流中插入 VABS 指令或等效的代码序列。通过使用此内在函数，可以从 C 或 C++ 代码中获取双精度浮点值的绝对值。

注意

__fabs 内在函数类似于标准 C 库函数 fabs。它在以下方面不同于标准库函数：可确保在基于 ARM 体系结构且配备 VFP 协处理器的处理器上将 __fabs 调用编译为单个内联机器指令。

语法

```
double __fabs(double val)
```

其中：

val 是一个双精度浮点值。

返回值

__fabs 内在函数以 **double** 形式返回 *val* 的绝对值。

另请参阅

- *__fabsf*
- *《汇编器指南》*中第5-98 页的*VABS、 VNEG 和 VSQRT*。

4.7.12 __fabsf

此内在函数是 __fabs 内在函数的单精度版本。它与 __fabs 的功能相当，只不过：

- 它使用 **float** 类型的自变量，而不是 **double** 类型的自变量
- 它返回 **float** 值，而不是 **double** 值。

另请参阅

- *__fabs*
- *《汇编器指南》*中第5-59 页的*V{Q}ABS 和 V{Q}NEG*。

4.7.13 __force_stores

此内在函数导致将在当前函数外部可见的所有变量写回到内存中 （如果它们已更改），如将其指针传递到该函数或从该函数中传出的变量。

此内在函数还用作调度屏障。

语法

```
void __force_stores(void)
```

另请参阅

- 第4-83 页的*__memory_changed*
- 第4-91 页的*__schedule_barrier*

4.7.14 __ldrex

此内在函数在编译器生成的指令流中插入 LDREX[*size*] 格式的指令。它允许在 C 或 C++ 代码中使用 LDREX 指令从内存中加载数据。LDREX[*size*] 中的 *size* 是 B （代表字节存储）或 H （代表半字存储）。如果没有指定大小，则执行字存储。

语法

```
unsigned int __ldrex(volatile void *ptr)
```

其中：

ptr 指向要从内存中加载的数据的地址。若要指定要加载的数据类型，请将参数类型转换为相应的指针类型。

表 4-8 __ldrex 内在函数支持的访问宽度

指令	加载的数据大小	C 类型转换
LDREXB	无符号字节	(**unsigned char** *)
LDREXB	有符号字节	(**signed char** *)
LDREXH	无符号半字	(**unsigned short** *)
LDREXH	有符号半字	(**short** *)
LDREX	字	(**int** *)

返回值

__ldrex 内在函数返回从 *ptr* 指向的内存地址中加载的数据。

错误

为不支持 LDREX 指令的目标进行编译时，编译器无法识别 __ldrex 内在函数。在这种情况下，编译器将生成警告或错误。

__ldrex 内在函数不支持对双字数据的访问。如果指定了不支持的访问宽度，编译器将生成错误。

示例

```
int foo(void)
{
    int loc = 0xff;
    return __ldrex((volatile char *)loc);
}
```

使用命令行选项 --cpu=6k 编译此代码时，将生成以下内容：

```
||foo|| PROC
    MOV      r0,#0xff
    LDREXB   r0,[r0]
    BX       lr
    ENDP
```

另请参阅

- *__ldrexd*
- 第4-95 页的*__strex*
- 第4-97 页的*__strexd*
- *《汇编器指南》*中第4-34 页的*LDREX 和 STREX*。

4.7.15 __ldrexd

此内在函数在编译器生成的指令流中插入 LDREXD 指令。它允许在 C 或 C++ 代码中使用 LDREXD 指令从内存中加载数据。它支持对双字数据的访问。

语法

```
unsigned long long __ldrexd(volatile void *ptr)
```

其中：

ptr 指向要从内存中加载的数据的地址。若要指定要加载的数据类型，请将参数类型转换为相应的指针类型。

表 4-9 __ldrex 内在函数支持的访问宽度

指令	加载的数据大小	C 类型转换
LDREXD	无符号 long long	(**unsigned long long** *)
LDREXD	有符号 long long	(**signed long long** *)

返回值

__ldrexd 内在函数返回从 *ptr* 指向的内存地址中加载的数据。

错误

为不支持 LDREXD 指令的目标进行编译时，编译器无法识别 __ldrexd 内在函数。在这种情况下，编译器将生成警告或错误。

__ldrexd 内在函数仅支持对双字数据的访问。如果指定了不支持的访问宽度，编译器将生成错误。

另请参阅

- 第4-79 页的*__ldrex*
- 第4-95 页的*__strex*
- 第4-97 页的*__strexd*
- *《汇编器指南》*中第4-34 页的*LDREX 和 STREX*。

4.7.16 __ldrt

此内在函数在编译器生成的指令流中插入 LDR{size}T 格式的汇编语言指令。它允许在 C 或 C++ 代码中使用 LDRT 指令从内存中加载数据。

语法

```
unsigned int __ldrt(const volatile void *ptr)
```

其中：

ptr 指向要从内存中加载的数据的地址。要指定所加载的数据大小，请将参数类型转换为相应的整型。

表 4-10 __ldrt 内在函数支持的访问宽度

指令[a]	加载的数据大小	C 类型转换
LDRSBT	有符号字节	(**signed char** *)
LDRBT	无符号字节	(**char** *)
LDRSHT	有符号半字	(**signed short int** *)
LDRHT	无符号半字	(**short int** *)
LDRT	字	(**int** *)

a. 或等效。

返回值

__ldrt 内在函数返回从 *ptr* 指向的内存地址中加载的数据。

错误

为不支持 LDRT 指令的目标进行编译时，编译器无法识别 __ldrt 内在函数。在这种情况下，编译器将生成警告或错误。

__ldrt 内在函数不支持对双字数据的访问。如果指定了不支持的访问宽度，编译器将生成错误。

示例

```
int foo(void)
{
    int loc = 0xff;
    return __ldrt((const volatile int *)loc);
}
```

使用缺省选项编译此代码时，将生成以下内容：

```
||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
```

另请参阅

- 第2-119 页的*--thumb*
- *《汇编器指南》*中第4-17 页的*LDR 和 STR （用户模式）*。

4.7.17 __memory_changed

此内在函数导致将在当前函数外部可见的所有变量写回到内存中 （如果它们已更改），然后从内存中重新读取，如将其指针传递到该函数或从该函数中传出的变量。

此内在函数还用作调度屏障。

语法

```
void __memory_changed(void)
```

另请参阅

- 第4-79 页的*__force_stores*
- 第4-91 页的*__schedule_barrier*

4.7.18 __nop

此内在函数在编译器生成的指令流中插入 NOP 指令或等效的代码序列。将为源代码中的每个 __nop 内在函数生成一个 NOP 指令。

除了正常删除不会到达的代码之外，编译器不会优化删除 NOP 指令。__nop 内在函数还用作编译器中的指令调度屏障。即，不会由于优化而将指令从 NOP 一侧移到另一侧。

注意

可以使用 __schedule_barrier 内在函数插入调度屏障，而无需生成 NOP 指令。

语法

```
void __nop(void)
```

另请参阅

- 第4-93 页的*__sev*
- 第4-91 页的*__schedule_barrier*
- 第4-101 页的*__wfe*
- 第4-102 页的*__wfi*
- 第4-102 页的*__yield*
- *《汇编器指南》*中第4-138 页的*NOP*、 *SEV*、 *WFE*、 *WFI 和 YIELD*
- *《编译器用户指南》*中第4-3 页的*通用内在函数*。

4.7.19 __pld

此内在函数在编译器生成的指令流中插入数据预取，例如 PLD。它允许从 C 或 C++ 程序中发信号通知内存系统，此后不久可能会从某个地址中加载数据。

语法

```
void __pld(...)
```

其中：

...　　表示任意数量的指针或整数自变量，用于指定要预取的内存地址。

限制

如果目标体系结构不支持数据预取，则此内在函数无效。

示例

```
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;
}
```

另请参阅

- *__pldw*
- 第4-86 页的*__pli*
- *《汇编器指南》*中第4-23 页的*PLD*、*PLDW 和 PLI*。

4.7.20 __pldw

此内在函数在编译器生成的指令流中插入 PLDW 指令。它允许从 C 或 C++ 程序中发信号通知内存系统，此后不久可能会从某个要写入的地址中加载数据。

语法

```
void __pldw(...)
```

其中：

...　　表示任意数量的指针或整数自变量，用于指定要预取的内存地址。

限制

如果目标体系结构不支持数据预取，则此内在函数无效。

此内在函数只在提供多重处理扩展的 ARMv7 及更高版本的体系结构中有效。即，在定义了预定义的宏 __TARGET_FEATURE_MULTIPROCESSING 时。

示例

```
void foo(int *bar)
{
     __pldw(bar);
}
```

另请参阅

- 第4-115 页的*编译器预定义*
- 第4-84 页的*__pld*
- 第4-86 页的*__pli*
- *《汇编器指南》*中第4-23 页的*PLD*、*PLDW 和 PLI*。

4.7.21 __pli

此内在函数在编译器生成的指令流中插入指令预取，例如 PLI。它允许从 C 或 C++ 程序中发信号通知内存系统，此后不久可能会从某个地址中加载指令。

语法

```
void __pli(...)
```

其中：

...　　表示任意数量的指针或整数自变量，用于指定要预取的指令地址。

限制

如果目标体系结构不支持指令预取，则此内在函数无效。

另请参阅

- 第4-84 页的*__pld*
- 第4-85 页的*__pldw*
- *《汇编器指南》*中第4-23 页的*PLD、 PLDW 和 PLI*。

4.7.22 __promise

此内在函数向编译器保证给定的表达式是非零的。这就允许编译器在向量化代码时执行更积极的优化。

语法

```
void __promise(expr)
```

其中， *expr* 是一个具有非零值的表达式。

另请参阅

- *《编译器用户指南》*中第3-16 页的*使用 __promise 改善向量化性能*。

4.7.23 __qadd

此内在函数在编译器生成的指令流中插入 QADD 指令或等效的代码序列。它允许从 C 或 C++ 代码中获取两个整数的饱和相加结果。

语法

```
int __qadd(int val1, int val2)
```

其中：

val1 是饱和相加运算的第一个被加数

val2 是饱和相加运算的第二个被加数。

返回值

__qadd 内在函数返回 *val1* 和 *val2* 的饱和相加结果。

另请参阅

- *__qdbl*
- 第4-88 页的*__qsub*
- *《汇编器指南》*中第4-90 页的*QADD*、 *QSUB*、 *QDADD 和 QDSUB*。

4.7.24 __qdbl

此内在函数在编译器生成的指令流中插入等效于整数与其本身的饱和相加的指令。它允许从 C 或 C++ 代码中获取整数的饱和加倍结果。

语法

```
int __qdbl(int val)
```

其中：

val 是要加倍的数据值。

返回值

__qdbl 内在函数返回 *val* 与其自身的饱和相加结果，或等效于 __qadd(*val*, *val*)。

另请参阅

- *__qadd*

4.7.25 __qsub

此内在函数在编译器生成的指令流中插入 QSUB 指令或等效的代码序列。它允许从 C 或 C++ 代码中获取两个整数的饱和相减结果。

语法

```
int __qsub(int val1, int val2)
```

其中：

val1 是饱和相减运算的被减数

val2 是饱和相减运算的减数。

返回值

__qsub 内在函数返回 *val1* 和 *val2* 的饱和相减结果。

另请参阅

- 第4-87 页的*__qadd*
- *《汇编器指南》*中第4-90 页的*QADD*、 *QSUB*、 *QDADD 和 QDSUB*。

4.7.26 __rbit

此内在函数在编译器生成的指令流中插入 RBIT 指令。它允许从 C 或 C++ 代码中颠倒 32 位字中的位顺序。

语法

```
unsigned int __rbit(unsigned int val)
```

其中：

val 是要颠倒位顺序的数据值。

返回值

__rbit 内在函数通过颠倒 val 位顺序返回从中获取的值。

另请参阅

- *《汇编器指南》*中第4-63 页的*REV*、 *REV16*、 *REVSH 和 RBIT*。

4.7.27 __rev

此内在函数在编译器生成的指令流中插入 REV 指令或等效的代码序列。它允许从 C 或 C++ 代码中将 32 位大端数据值转换为小端数据值，或者将 32 位小端数据值转换为大端数据值。

注意

当编译器识别某些表达式后，将自动引入 REV。

语法

```
unsigned int __rev(unsigned int val)
```

其中：

val 是一个 **unsigned int**。

返回值

__rev 内在函数通过颠倒 val 字节顺序返回从中获取的值。

另请参阅

- *《汇编器指南》*中第4-63 页的*REV*、 *REV16*、 *REVSH 和 RBIT*。

4.7.28 __return_address

此内在函数返回当前函数的返回地址。

语法

```
unsigned int __return_address(void)
```

返回值

__return_address 内在函数返回在从当前函数返回时使用的链接寄存器的值。

限制

__return_address 内在函数*不*影响编译器执行优化的功能，如内联、尾调用和代码共享。如果进行了优化，__return_address 返回的值将反映所执行的优化：

不优化 如果未执行任何优化，__return_address 从函数 foo 中返回的值是 foo 的返回地址。

内联优化 如果函数 foo 已内联到函数 bar，则 __return_address 从 foo 中返回的值是 bar 的返回地址。

尾调用优化 如果从函数 bar 中尾调用函数 foo，则 __return_address 从 foo 中返回的值是 bar 的返回地址。

另请参阅

- 第4-114 页的*其他内置函数*
- 第4-74 页的*__current_pc*
- 第4-74 页的*__current_sp*
- *《编译器用户指南》*中第7-25 页的*访问 sp、 lr 或 pc 的旧内联汇编器*

4.7.29 __ror

此内在函数在编译器生成的指令流中插入 ROR 指令或操作数循环移位。它允许从 C 或 C++ 代码中将值向右循环移指定的位数。

注意

当编译器识别某些表达式后，将自动引入 ROR。

语法

```
unsigned int __ror(unsigned int val, unsigned int shift)
```

其中：

val 是要向右移的值

shift 是在范围 1-31 内移位的常数。

返回值

__ror 内在函数返回向右循环移 shift 位的 val 值。

另请参阅

- *《汇编器指南》*中第4-65 页的*ASR*、 *LSL*、 *LSR*、 *ROR* 和 *RRX*。

4.7.30 __schedule_barrier

此内在函数创建一个序列点，编译器没有合并序列点前后的运算。调度屏障不会导致更新内存。如果变量保存在寄存器中，则会对其进行原位更新，而不会将其写出。

此内在函数类似于 __nop 内在函数，只不过不生成 NOP 指令。

语法

```
void schedule_barrier(void)
```

另请参阅

- 第4-83 页的*__nop*

4.7.31 __semihost

此内在函数在编译器生成的指令流中插入 SVC 或 BKPT 指令。它允许从 C 或 C++ 中进行与目标体系结构无关的半主机调用。

语法

```
int __semihost(int val, const void *ptr)
```

其中：

val 是半主机请求的请求代码。
有关详细信息，请参阅 *《开发指南》*中的第 8 章 *半主机*。

ptr 是一个指向参数/结果块的指针。
有关详细信息，请参阅 *《开发指南》*中的第 8 章 *半主机*。

返回值

有关半主机调用结果的详细信息，请参阅 *《开发指南》*中的第 8 章 *半主机*。

用法

可以从 C 或 C++ 中使用此内在函数为目标和指令集生成相应的半主机调用：

`SVC 0x123456`	对于所有体系结构，在 ARM 状态下。
`SVC 0xAB`	在 Thumb 状态下，不包括 ARMv7-M。不能保证来自 ARM 或第三方的*所有*调试目标上均会出现这种行为。
`BKPT 0xAB`	对于 ARMv7-M，仅限 Thumb-2。

限制

ARMv7 之前的 ARM 处理器使用 `SVC` 指令进行半主机调用。不过，如果为 Cortex M-profile 处理器进行编译，则会使用 `BKPT` 指令实现半主机。

示例

```
char buffer[100];
...
void foo(void)
{
    __semihost(0x01, (const void *)buf); // equivalent in thumb state to
                                         // int __svc(0xAB) my_svc(int, int *);
                                         // result = my_svc(0x1, &buffer);
}
```

使用选项 `--thumb` 编译此代码时，将生成以下内容：

```
||foo|| PROC
    ...
    LDR      r1,|L1.12|
    MOVS     r0,#1
    SVC      #0xab
    ...
|L1.12|
    ...
buffer
    %        400
```

另请参阅

- 第2-30 页的*--cpu=list*
- 第2-119 页的*--thumb*
- 第4-16 页的*__svc*
- *《汇编器指南》*中第4-128 页的*BKPT*
- *《汇编器指南》*中第4-129 页的*SVC*
- *《开发指南》*中的第 8 章 *半主机*。

4.7.32 __sev

此内在函数在编译器生成的指令流中插入 SEV 指令。

语法

```
void __sev(void)
```

错误

为不支持 SEV 指令的目标进行编译时，编译器无法识别 __sev 内在函数。在这种情况下，编译器将生成警告或错误。

另请参阅

- 第4-83 页的*__nop*
- 第4-101 页的*__wfe*
- 第4-102 页的*__wfi*
- 第4-102 页的*__yield*
- *《汇编器指南》*中第4-138 页的*NOP*、 *SEV*、 *WFE*、 *WFI 和 YIELD*。

4.7.33 __sqrt

此内在函数在编译器生成的指令流中插入 VFP VSQRT 指令。它允许从 C 或 C++ 代码中获取双精度浮点值的平方根。

注意

__sqrt 内在函数类似于标准 C 库函数 sqrt。它在以下方面不同于标准库函数：可确保在基于 ARM 体系结构且配备 VFP 协处理器的处理器上将 __sqrt 调用编译为单个内联机器指令。

语法

```
double __sqrt(double val)
```

其中：

val 是一个双精度浮点值。

返回值

__sqrt 内在函数以 **double** 形式返回 *val* 的平方根。

错误

为没有配备 VFP 协处理器的目标进行编译时，编译器无法识别 __sqrt 内在函数。在这种情况下，编译器将生成警告或错误。

另请参阅

- *__sqrtf*
- *《汇编器指南》*中第5-98 页的*VABS*、 *VNEG 和 VSQRT*。

4.7.34 __sqrtf

此内在函数是 __sqrtf 内在函数的单精度版本。它与 __sqrt 的功能相当，只不过：

- 它使用 **float** 类型的自变量，而不是 **double** 类型的自变量
- 它返回 **float** 值，而不是 **double** 值。

另请参阅

- 第4-93 页的*__sqrt*
- *《汇编器指南》*中第5-98 页的*VABS*、 *VNEG 和 VSQRT*。

4.7.35 __ssat

此内在函数在编译器生成的指令流中插入 SSAT 指令。它允许从 C 或 C++ 代码中饱和有符号值。

语法

```
int __ssat(int val, unsigned int sat)
```

其中：

val 是要饱和的值。

sat 是要饱和到的位位置。
sat 必须在 1 到 32 之间。

返回值

__ssat 内在函数返回饱和到有符号范围 $-2^{sat-1} \le x \le 2^{sat-1} - 1$ 的 *val*。

错误

为不支持 SSAT 指令的目标进行编译时，编译器无法识别 __ssat 内在函数。在这种情况下，编译器将生成警告或错误。

另请参阅

- 第4-100 页的*__usat*
- *《汇编器指南》*中第4-92 页的*SSAT 和 USAT*。

4.7.36 __strex

此内在函数在编译器生成的指令流中插入 STREX[size] 格式的指令。它允许在 C 或 C++ 代码中使用 STREX 指令将数据存储到内存中。

语法

```
int __strex(unsigned int val, volatile void *ptr)
```

其中：

val 是要写入到内存中的值。

ptr 指向要写入到内存中的数据的地址。若要指定要写入的数据的大小，请将参数类型转换为相应的整型。

表 4-11 __strex 内在函数支持的访问宽度

指令	存储的数据大小	C 类型转换
STREXB	无符号字节	(**char** *)
STREXH	无符号半字	(**short** *)
STREX	字	(**int** *)

返回值

__strex 内在函数返回以下内容：

0 如果 STREX 指令执行成功

1 如果 STREX 指令已锁定。

错误

为不支持 STREX 指令的目标进行编译时，编译器无法识别 __strex 内在函数。在这种情况下，编译器将生成警告或错误。

__strex 内在函数不支持对双字数据的访问。如果指定了不支持的访问宽度，编译器将生成错误。

示例

```
int foo(void)
{
    int loc=0xff;
    return(!__strex(0x20, (volatile char *)loc));
}
```

使用命令行选项 --cpu=6k 编译此代码时，将生成以下内容：

```
||foo|| PROC
    MOV      r0,#0xff
    MOV      r2,#0x20
    STREXB   r1,r2,[r0]
    RSBS     r0,r1,#1
    MOVCC    r0,#0
    BX       lr
ENDP
```

另请参阅

- 第4-79 页的*__ldrex*
- 第4-80 页的*__ldrexd*
- *__strexd*
- *《汇编器指南》*中第4-34 页的*LDREX 和 STREX*。

4.7.37 __strexd

此内在函数在编译器生成的指令流中插入 STREXD 指令。它允许在 C 或 C++ 代码中使用 STREXD 指令将数据存储到内存中。它支持双字数据到内存的独占存储。

语法

```
int __strexd(unsigned long long val, volatile void *ptr)
```

其中：

val　　是要写入到内存中的值。

ptr　　指向要写入到内存中的数据的地址。若要指定要写入的数据的大小，请将参数类型转换为相应的整型。

表 4-12 __strexd 内在函数支持的访问宽度

指令	存储的数据大小	C 类型转换
STREXD	无符号 long long	(**unsigned long long** *)
STREXD	有符号 long long	(**signed long long** *)

返回值

__strexd 内在函数返回以下内容：

0　　如果 STREXD 指令执行成功

1　　如果 STREXD 指令已锁定。

错误

为不支持 STREXD 指令的目标进行编译时，编译器无法识别 __strexd 内在函数。在这种情况下，编译器将生成警告或错误。

__strexd 内在函数仅支持对双字数据的访问。如果指定了不支持的访问宽度，编译器将生成错误。

另请参阅

- 第4-79 页的*__ldrex*
- 第4-80 页的*__ldrexd*
- 第4-95 页的*__strex*
- *《汇编器指南》* 中第4-34 页的*LDREX 和 STREX*。

4.7.38 __strt

此内在函数在编译器生成的指令流中插入 STR{size}T 格式的汇编语言指令。它允许在 C 或 C++ 代码中使用 STRT 指令将数据存储到内存中。

语法

```
void __strt(unsigned int val, volatile void *ptr)
```

其中：

val 是要写入到内存中的值。

ptr 指向要写入到内存中的数据的地址。若要指定要写入的数据的大小，请将参数类型转换为相应的整型。

表 4-13 __strt 内在函数支持的访问宽度

指令	加载的数据大小	C 类型转换
STRBT	无符号字节	(**char** *)
STRHT	无符号半字	(**short int** *)
STRT	字	(**int** *)

错误

为不支持 STRT 指令的目标进行编译时，编译器无法识别 __strt 内在函数。在这种情况下，编译器将生成警告或错误。

__strt 内在函数不支持对有符号数据或双字数据的访问。如果指定了不支持的访问宽度，编译器将生成错误。

示例

```
void foo(void)
{
    int loc=0xff;
    __strt(0x20, (volatile char *)loc);
}
```

编译此代码时，将生成以下内容：

```
||foo|| PROC
    MOV      r0,#0xff
    MOV      r1,#0x20
    STRBT    r1,[r0],#0
    BX       lr
    ENDP
```

另请参阅

- 第2-119 页的*--thumb*
- *《汇编器指南》*中第4-17 页的*LDR 和 STR （用户模式）*。

4.7.39 __swp

此内在函数在编译器生成的指令流中插入 SWP{size} 指令。它允许从 C 或 C++ 代码中在内存位置之间交换数据。

注意

在 ARMv6 和更高版本中，不提倡使用 SWP 和 SWPB。

语法

```
unsigned int __swp(unsigned int val, volatile void *ptr)
```

其中：

val 是要写入到内存中的数据值。

ptr 指向要写入到内存中的数据的地址。若要指定要写入的数据的大小，请将参数类型转换为相应的整型。

表 4-14 __swp 内在函数支持的访问宽度

指令	加载的数据大小	C 类型转换
SWPB	无符号字节	(**char** *)
SWP	字	(**int** *)

返回值

在 *val* 覆盖以前位于 *ptr* 所指向的内存地址中的数据值之前， __swp 内在函数将返回该值。

示例

```
int foo(void)
{
    int loc=0xff;
    return(__swp(0x20, (volatile int *)loc));
}
```

编译此代码时，将生成以下内容：

```
||foo|| PROC
    MOV      r1, #0xff
    MOV      r0, #0x20
    SWP      r0, r0, [r1]
    BX       lr
    ENDP
```

另请参阅

- *《汇编器指南》*中第4-38 页的*SWP 和 SWPB*。

4.7.40 __usat

此内在函数在编译器生成的指令流中插入 USAT 指令。它允许从 C 或 C++ 代码中饱和无符号值。

语法

```
int __usat(unsigned int val, unsigned int sat)
```

其中：

val 是要饱和的值。

sat 是要饱和到的位位置。
usat 必须在 0 到 31 之间。

返回值

__usat 内在函数返回饱和到无符号范围 $0 \le x \le 2^{sat-1} - 1$ 的 *val*。

错误

为不支持 USAT 指令的目标进行编译时，编译器无法识别 __usat 内在函数。在这种情况下，编译器将生成警告或错误。

另请参阅

- 第4-94 页的*__ssat*
- *《汇编器指南》*中第4-92 页的*SSAT 和 USAT*。

4.7.41 __wfe

此内在函数在编译器生成的指令流中插入 WFE 指令。

在 v6T2 体系结构上，WFE 指令作为 NOP 指令进行执行。

语法

```
void __wfe(void)
```

错误

为不支持 WFE 指令的目标进行编译时，编译器无法识别 __wfe 内在函数。在这种情况下，编译器将生成警告或错误。

另请参阅

- 第4-102 页的*__wfi*
- 第4-83 页的*__nop*
- 第4-93 页的*__sev*
- 第4-102 页的*__yield*
- *《汇编器指南》*中第4-138 页的*NOP、 SEV、 WFE、 WFI 和 YIELD*。

4.7.42 __wfi

此内在函数在编译器生成的指令流中插入 WFI 指令。

在 v6T2 体系结构上，WFI 指令作为 NOP 指令进行执行。

语法

```
void __wfi(void)
```

错误

为不支持 WFI 指令的目标进行编译时，编译器无法识别 __wfi 内在函数。在这种情况下，编译器将生成警告或错误。

另请参阅

- *__yield*
- 第4-83 页的*__nop*
- 第4-93 页的*__sev*
- 第4-101 页的*__wfe*
- *《汇编器指南》*中第4-138 页的*NOP*、 *SEV*、 *WFE*、 *WFI 和 YIELD*。

4.7.43 __yield

此内在函数在编译器生成的指令流中插入 YIELD 指令。

语法

```
void __yield(void)
```

错误

为不支持 YIELD 指令的目标进行编译时，编译器无法识别 __yield 内在函数。在这种情况下，编译器将生成警告或错误。

另请参阅

- 第4-83 页的*__nop*
- 第4-93 页的*__sev*
- 第4-101 页的*__wfe*
- 第4-102 页的*__wfi*
- *《汇编器指南》*中第4-138 页的*NOP*、 *SEV*、 *WFE*、 *WFI 和 YIELD*。

4.7.44 ARMv6 SIMD 内在函数

ARM 体系结构 v6 指令集体系结构在 ARMv6 中添加了 60 多条 SIMD 指令，以便通过软件有效地实现高性能的媒体应用程序。

ARM 编译器支持映射到 ARMv6 SIMD 指令的内在函数。为 ARMv6 体系结构或处理器编译代码时，可以使用这些内在函数。以下列表给出了这些内在函数的函数原型。列表中给出的函数原型描述了内在函数实现的原始或基本形式的 ARMv6 指令。要获取某个内在函数实现的基本指令名称，请删除内在函数名称前面的下划线 (__)。例如， `__qadd16` 内在函数对应于 ARMv6 `QADD16` 指令。

——注意——

可以确保针对 ARM v6 体系结构或处理器将每个 ARMv6 SIMD 内在函数编译为单个内联机器指令。但是，编译器检测到需要执行此操作的场合时，可能会使用优化形式的基本指令。

ARMv6 SIMD 指令可以在*应用程序状态寄存器* (APSR) 中设置 `GE[3:0]` 位。SIMD 指令可以更新这些标记以指示 SIMD 运算的每个 8/16 位片的 “大于或等于”状态。

ARM 编译器将 `GE[3:0]` 位视为全局变量。要从 C 或 C++ 程序中访问这些位，请执行以下任一操作：

- 通过已命名的寄存器变量访问 APSR 的第 16-19 位
- 使用 `__sel` 内在函数控制 `SEL` 指令。

```
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 __uhsub16(unsigned int, unsigned int)
unsigned int __uhsub8(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)
```

另请参阅

- 第4-108 页的*已命名的寄存器变量*
- *《汇编器指南》*中第2-6 页的*寄存器*
- *《汇编器指南》*中第4-61 页的*SEL*
- *《汇编器指南》*中的第 5 章 *NEON 和 VFP 编程*。

4.7.45 ETSI 基本运算

RVCT 支持原始 ETSI 基本运算系列 （如 ETSI G.729 建议：《使用共轭结构代数码激励线性预测 (CS-ACELP) 的 8 Kb/s 语音编码》(*Coding of speech at 8 kbit/s using conjugate-structure algebraic-code-excited linear prediction (CS-ACELP)*) 所述）。

要在您自己的代码中使用 ETSI 基本运算，请包含标准头文件 dspfns.h。表 4-15 列出了 dspfns.h 中提供的内在函数。

表 4-15 RVCT 中支持的 ETSI 基本运算

内在函数				
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
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

头文件 dspfns.h 还将某些状态标记作为全局变量公开，以便在 C 或 C++ 程序中使用。表 4-16 列出了 dspfns.h 公开的状态标记。

表 4-16 RVCT 中公开的 ETSI 状态标记

状态标记	说明
Overflow	溢出状态标记。 通常，饱和函数对溢出具有粘着效果。
Carry	进位状态标记。

示例

```
#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
}
```

另请参阅

- 头文件 dspfns.h 中的 ETSI 基本运算定义，其形式为 C 代码和内在函数的组合
- *《编译器用户指南》*中第4-6 页的*ETSI 基本运算*
- ETSI 建议 G.191：《语音和音频编码标准化软件工具》 (*Software tools for speech and audio coding standardization*)
- *《ITU-T 软件工具库 2005 用户手册》*，收录为 ETSI 建议书 G.191 的一部分
- ETSI 建议 G723.1：《传输速率为 5.3 和 6.3 Kb/s 的多媒体通信双速率语音编码器》 (*Dual rate speech coder for multimedia communications transmitting at 5.3 and 6.3 kbit/s*)
- ETSI 建议 G.729：《使用共轭结构代数码激励线性预测 (CS-ACELP) 的 8 Kb/s 语音编码》 (*Coding of speech at 8 kbit/s using conjugate-structure algebraic-code-excited linear prediction (CS-ACELP)*)。

4.7.46 C55x 内在函数

ARM 编译器支持某些 TI C55x 编译器内在函数的仿真。

要在您自己的代码中使用 TI C55x 内在函数，请包含标准头文件 c55x.h。表 4-17 列出了 c55x.h 中提供的内在函数。

表 4-17 RVCT 中支持的 TI C55x 内在函数

内在函数			
_abss	_lshrs	_rnd	_smas
_count	_lsadd	_norm	_smpy
_divs	_lsmpy	_round	_sneg
_labss	_lsneg	_roundn	_sround
_lmax	_lsshl	_sadd	_sroundn
_lmin	_lssub	_shl	_sshl
_lnorm	_max	_shrs	_ssub
_lshl	_min	_smac	

示例

```
#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
}
```

另请参阅

- 头文件 c55x.h，以了解有关 ARM 的 C55x 内在函数实现的详细信息
- 德州仪器公司的网站 http://www.ti.com 上提供有关 TI 编译器内在函数信息的出版物。

4.7.47 已命名的寄存器变量

通过使用编译器，您可以使用已命名的寄存器变量访问基于 ARM 体系结构的处理器的寄存器。支持的已命名寄存器变量为：

- 位于文件范围内
- 位于局部范围内，但不在函数参数中。

语法

```
register type var-name __asm(reg);
```

其中：

type 是已命名寄存器变量的类型。

在已命名的寄存器变量声明中，可以使用与已命名的寄存器大小相同的任何类型。该类型可以为结构，但要注意位域布局区分端标记。

var-name 是已命名寄存器变量的名称。

reg 是一个字符串，表示基于 ARM 体系结构的处理器上的寄存器名称。

表 4-18 中显示了基于 ARM 体系结构的处理器上可用于已命名寄存器变量的寄存器。

表 4-18 基于 ARM 体系结构的处理器上提供的已命名寄存器

寄存器	用于 __asm 的字符串	处理器
CPSR	"cpsr" 或 "apsr"	所有处理器
BASEPRI	"basepri"	Cortex-M3
BASEPRI_MAX	"basepri_max"	Cortex-M3
CONTROL	"control"	Cortex-M1、 Cortex-M3
EAPSR	"eapsr"	Cortex-M1、 Cortex-M3

表 4-18 基于 ARM 体系结构的处理器上提供的已命名寄存器 （续）

寄存器	用于 __asm 的字符串	处理器
EPSR	"epsr"	Cortex-M1、 Cortex-M3
FAULTMASK	"faultmask"	Cortex-M3
IAPSR	"iapsr"	Cortex-M1、 Cortex-M3
IEPSR	"iepsr"	Cortex-M1、 Cortex-M3
IPSR	"ipsr"	Cortex-M1、 Cortex-M3
MSP	"msp"	Cortex-M1、 Cortex-M3
PRIMASK	"primask"	Cortex-M1、 Cortex-M3
PSP	"psp"	Cortex-M1、 Cortex-M3
r0 到 r12	"r0" 至 "r12"	所有处理器
r13 或 sp	"r13" 或 "sp"	所有处理器
r14 或 lr	"r14" or "lr"	所有处理器
r15 或 pc	"r15" or "pc"	所有处理器
SPSR	"spsr"	所有处理器
XPSR	"xpsr"	Cortex-M1、 Cortex-M3

在包含 VFP 的目标上，表 4-19 中的寄存器还可用于已命名的寄存器变量。

表 4-19 包含 VFP 的目标上提供的已命名寄存器

寄存器	用于 __asm 的字符串
FPSID	"fpsid"
FPSCR	"fpscr"
FPEXC	"fpexc"

示例

```
void foo(void)
{
    register int foo __asm("r0");
}
```

在此示例中，将 foo 声明为寄存器 r0 的已命名寄存器变量。

另请参阅

- *《编译器用户指南》*中第4-9 页的*已命名的寄存器变量*。

4.8 VFP 状态内在函数

编译器提供了用于读取*浮点状态和控制寄存器*(FPSCR)的内在函数。

注意

最好使用已命名寄存器变量的方法代替直接读取此寄存器。这种方法使得访问更为有效。请参阅第4-108 页的*已命名的寄存器变量*。

4.8.1 __vfp_status

此内在函数将读取 FPSCR。

语法

```
unsigned int __vfp_status(unsigned int mask, unsigned int flags);
```

错误

如果试图在为没有 VFP 的目标进行编译时使用此内在函数，编译器将生成错误。

另请参阅

- 有关 FPSCR 寄存器的信息，请参阅 *《ARM 体系结构参考手册》*

4.9 GNU 内置函数

这些函数提供了与 GNU 库头文件的兼容性。

4.9.1 非标准函数

__builtin_alloca(), __builtin_bcmp(), __builtin_exit(), __builtin_gamma(), __builtin_gammaf(), __builtin_gammal(), __builtin_index(), __builtin_rindex(), __builtin_strcasecmp(), __builtin_strncasecmp().

4.9.2 C99 函数

__builtin_Exit(), __builtin_acoshf(), __builtin_acoshl(), __builtin_acosh(), __builtin_asinhf(), __builtin_asinhl(), __builtin_asinh(), __builtin_atanhf(), __builtin_atanhl(), __builtin_atanh(), __builtin_cabsf(), __builtin_cabsl(), __builtin_cabs(), __builtin_cacosf(), __builtin_cacoshf(), __builtin_cacoshl(), __builtin_cacosh(), __builtin_cacosl(), __builtin_cacos(), __builtin_cargf(), __builtin_cargl(), __builtin_carg(), __builtin_casinf(), __builtin_casinhf(), __builtin_casinhl(), __builtin_casinh(), __builtin_casinl(), __builtin_casin(), __builtin_catanf(), __builtin_catanhf(), __builtin_catanhl(), __builtin_catanh(), __builtin_catanl(), __builtin_catan(), __builtin_cbrtf(), __builtin_cbrtl(), __builtin_cbrt(), __builtin_ccosf(), __builtin_ccoshf(), __builtin_ccoshl(), __builtin_ccosh(), __builtin_ccosl(), __builtin_ccos(), __builtin_cexpf(), __builtin_cexpl(), __builtin_cexp(), __builtin_cimagf(), __builtin_cimagl(), __builtin_cimag(), __builtin_clogf(), __builtin_clogl(), __builtin_clog(), __builtin_conjf(), __builtin_conjl(), __builtin_conj(), __builtin_copysignf(), __builtin_copysignl(), __builtin_copysign(), __builtin_cpowf(), __builtin_cpowl(), __builtin_cpow(), __builtin_cprojf(), __builtin_cprojl(), __builtin_cproj(), __builtin_crealf(), __builtin_creall(), __builtin_creal(), __builtin_csinf(), __builtin_csinhf(), __builtin_csinhl(), __builtin_csinh(), __builtin_csinl(), __builtin_csin(), __builtin_csqrtf(), __builtin_csqrtl(), __builtin_csqrt(), __builtin_ctanf(), __builtin_ctanhf(), __builtin_ctanhl(), __builtin_ctanh(), __builtin_ctanl(), __builtin_ctan(), __builtin_erfcf(), __builtin_erfcl(), __builtin_erfc(), __builtin_erff(), __builtin_erfl(), __builtin_erf(), __builtin_exp2f(), __builtin_exp2l(), __builtin_exp2(), __builtin_expm1f(), __builtin_expm1l(), __builtin_expm1(), __builtin_fdimf(), __builtin_fdiml(), __builtin_fdim(), __builtin_fmaf(), __builtin_fmal(), __builtin_fmaxf(), __builtin_fmaxl(), __builtin_fmax(), __builtin_fma(), __builtin_fminf(), __builtin_fminl(), __builtin_fmin(), __builtin_hypotf(), __builtin_hypotl(), __builtin_hypot(), __builtin_ilogbf(), __builtin_ilogbl(), __builtin_ilogb(), __builtin_imaxabs(), __builtin_isblank(), __builtin_isfinite(), __builtin_isinf(), __builtin_isnan(), __builtin_isnanf(), __builtin_isnanl(), __builtin_isnormal(), __builtin_iswblank(), __builtin_lgammaf(), __builtin_lgammal(), __builtin_lgamma(), __builtin_llabs(), __builtin_llrintf(), __builtin_llrintl(), __builtin_llrint(), __builtin_llroundf(), __builtin_llroundl(), __builtin_llround(), __builtin_log1pf(), __builtin_log1pl(), __builtin_log1p(), __builtin_log2f(), __builtin_log2l(), __builtin_log2(), __builtin_logbf(), __builtin_logbl(),

__builtin_logb(), __builtin_lrintf(), __builtin_lrintl(), __builtin_lrint(),
__builtin_lroundf(), __builtin_lroundl(), __builtin_lround(),
__builtin_nearbyintf(), __builtin_nearbyintl(), __builtin_nearbyint(),
__builtin_nextafterf(), __builtin_nextafterl(), __builtin_nextafter(),
__builtin_nexttowardf(), __builtin_nexttowardl(), __builtin_nexttoward(),
__builtin_remainderf(), __builtin_remainderl(), __builtin_remainder(),
__builtin_remquof(), __builtin_remquol(), __builtin_remquo(), __builtin_rintf(),
__builtin_rintl(), __builtin_rint(), __builtin_roundf(), __builtin_roundl(),
__builtin_round(), __builtin_scalblnf(), __builtin_scalblnl(),
__builtin_scalbln(), __builtin_scalbnf(), __builtin_calbnl(),
__builtin_scalbn(), __builtin_signbit(), __builtin_signbitf(),
__builtin_signbitl(), __builtin_snprintf(), __builtin_tgammaf(),
__builtin_tgammal(), __builtin_tgamma(), __builtin_truncf(), __builtin_truncl(),
__builtin_trunc(), __builtin_vfscanf(), __builtin_vscanf(),
__builtin_vsnprintf(), __builtin_vsscanf().

4.9.3 C90 保留命名空间中的 C99 函数

__builtin_acosf(), __builtin_acosl(), __builtin_asinf(), __builtin_asinl(),
__builtin_atan2f(), __builtin_atan2l(), __builtin_atanf(), __builtin_atanl(),
__builtin_ceilf(), __builtin_ceill(), __builtin_cosf(), __builtin_coshf(),
__builtin_coshl(), __builtin_cosl(), __builtin_expf(), __builtin_expl(),
__builtin_fabsf(), __builtin_fabsl(), __builtin_floorf(), __builtin_floorl(),
__builtin_fmodf(), __builtin_fmodl(), __builtin_frexpf(), __builtin_frexpl(),
__builtin_ldexpf(), __builtin_ldexpl(), __builtin_log10f(), __builtin_log10l(),
__builtin_logf(), __builtin_logl(), __builtin_modfl(), __builtin_modf(),
__builtin_powf(), __builtin_powl(), __builtin_sinf(), __builtin_sinhf(),
__builtin_sinhl(), __builtin_sinl(), __builtin_sqrtf(), sqrtl, __builtin_tanf(),
__builtin_tanhf(), __builtin_tanhl(), __builtin_tanl().

4.9.4 C94 函数

__builtin_swalnum(), __builtin_iswalpha(), __builtin_iswcntrl(),
__builtin_iswdigit(), __builtin_iswgraph(), __builtin_iswlower(),
__builtin_iswprint(), __builtin_iswpunct(), __builtin_iswspace(),
__builtin_iswupper(), __builtin_iswxdigit(), __builtin_towlower(),
__builtin_towupper().

4.9.5 C90 函数

__builtin_abort(), __builtin_abs(), __builtin_acos(), __builtin_asin(),
__builtin_atan2(), __builtin_atan(), __builtin_calloc(), __builtin_ceil(),
__builtin_cosh(), __builtin_cos(), __builtin_exit(), __builtin_exp(),
__builtin_fabs(), __builtin_floor(), __builtin_fmod(), __builtin_fprintf(),
__builtin_fputc(), __builtin_fputs(), __builtin_frexp(), __builtin_fscanf(),
__builtin_isalnum(), __builtin_isalpha(), __builtin_iscntrl(),
__builtin_isdigit(), __builtin_isgraph(), __builtin_islower(),
__builtin_isprint(), __builtin_ispunct(), __builtin_isspace(),

__builtin_isupper(), __builtin_isxdigit(), __builtin_tolower(),
__builtin_toupper(), __builtin_labs(), __builtin_ldexp(), __builtin_log10(),
__builtin_log(), __builtin_malloc(), __builtin_memchr(), __builtin_memcmp(),
__builtin_memcpy(), __builtin_memset(), __builtin_modf(), __builtin_pow(),
__builtin_printf(), __builtin_putchar(), __builtin_puts(), __builtin_scanf(),
__builtin_sinh(), __builtin_sin(), __builtin_snprintf(), __builtin_sprintf(),
__builtin_sqrt(), __builtin_sscanf(), __builtin_strcat(), __builtin_strchr(),
__builtin_strcmp(), __builtin_strcpy(), __builtin_strcspn(),
__builtin_strlen(), __builtin_strncat(), __builtin_strncmp(),
__builtin_strncpy(), __builtin_strpbrk(), __builtin_strrchr(),
__builtin_strspn(), __builtin_strstr(), __builtin_tanh(), __builtin_tan(),
__builtin_vfprintf(), __builtin_vprintf(), __builtin_vsprintf().

4.9.6 C99 浮点函数

__builtin_huge_val(), __builtin_huge_valf(), __builtin_huge_vall(),
__builtin_inf(), __builtin_nan(), __builtin_nanf(), __builtin_nanl(),
__builtin_nans(), __builtin_nansf(), __builtin_nansl().

4.9.7 其他内置函数

__builtin_clz(), __builtin_constant_p(), __builtin_ctz(), builtin_ctzl(),
__builtin_ctzll(), __builtin_expect(), __builtin_ffs(), __builtin_ffsl(),
__builtin_ffsll(), __builtin_frame_address(), __builtin_return_address(),
__builtin_popcount(), __builtin_signbit().

4.10 编译器预定义

本节介绍 ARM 编译器的预定义宏。

4.10.1 预定义宏

表 4-20 列出了 ARM 编译器为 C 和 C++ 预定义的宏名称。如果值字段为空，则表明只定义了符号。

表 4-20 预定义宏

名称	值	定义时间
__arm__	-	始终为 ARM 编译器定义，即使指定了 --thumb 选项。 另请参阅 __ARMCC_VERSION
__ARMCC_VERSION	*ver*	始终定义。它是一个十进制数字，可确保随着版本的更新而增加。格式为 *PVbbbb*，其中： • *P* 是主版本 • *V* 是次版本 • *bbbb* 是内部版本号。 **注意** 可以使用此编号区别 RVCT 和用于定义 __arm__ 的其他工具。
__APCS_INTERWORK	-	指定 --apcs /interwork 选项或将 CPU 体系结构设置为 ARMv5T 或更高版本时。
__APCS_ROPI	-	指定 --apcs /ropi 选项时。
__APCS_RWPI	-	指定 --apcs /rwpi 选项时。
__APCS_FPIC	-	指定 --apcs /fpic 选项时。
__ARRAY_OPERATORS	-	在 C++ 编译器模式下指定启用数组新建和删除。
__BASE_FILE__	*name*	始终定义。类似于 __FILE__，但指示的是主源文件而不是当前文件 （即，当前文件是所包含的文件时）。
__BIG_ENDIAN	-	如果为大端目标进行编译。
_BOOL	-	在 C++ 编译器模式下，将 **bool** 指定为关键字。
__cplusplus	-	在 C++ 编译器模式下。
__CC_ARM	1	对于 ARM 编译器，始终设置为 1，即使指定了 --thumb 选项。

表 4-20 预定义宏 （续）

名称	值	定义时间
__CHAR_UNSIGNED__	–	在 GNU 模式下。当且仅当 **char** 为无符号类型时，才会对其进行定义。
__DATE__	*date*	始终定义。
__EDG__	–	始终定义。
__EDG_IMPLICIT_USING_STD	–	在 C++ 模式下指定 --implicit_using_std 选项时。
__EDG_VERSION__	–	始终设置为整型值，它表示 *Edison Design Group* (EDG) 前端的版本号。例如，将 3.8 版表示为 308。 *EDG 前端的版本号不必与 RVCT 或 RealView Development Suite 版本号相匹配。*
__EXCEPTIONS	1	在 C++ 模式下指定 --exceptions 选项时。
__FEATURE_SIGNED_CHAR	–	指定 --signed_chars 选项 （由 CHAR_MIN 和 CHAR_MAX 使用）时。
__FILE__	*name*	始终定义为字符串文字。
__FP_FAST	–	指定 --fpmode=fast 选项时。
__FP_FENV_EXCEPTIONS	–	指定 --fpmode=ieee_full 或 --fpmode=ieee_fixed 选项时。
__FP_FENV_ROUNDING	–	指定 --fpmode=ieee_full 选项时。
__FP_IEEE	–	指定 --fpmode=ieee_full、--fpmode=ieee_fixed 或 --fpmode=ieee_no_fenv 选项时。
__FP_INEXACT_EXCEPTION	–	指定 --fpmode=ieee_full 选项时。
__GNUC__	*ver*	指定 --gnu 选项时。它是一个整数，用于显示所使用的 GNU 模式的当前主版本。
__GNUC_MINOR__	*ver*	指定 --gnu 选项时。它是一个整数，用于显示所使用的 GNU 模式的当前次版本。
__GNUG__	*ver*	在 GNU 模式下指定 --cpp 选项时。它具有与 __GNUC__ 相同的值。
__IMPLICIT_INCLUDE	–	指定 --implicit_include 选项时。
__INTMAX_TYPE__	–	在 GNU 模式下。它为 intmax_t **typedef** 定义正确的基础类型。
__LINE__	*num*	始终设置。它是包含此宏的源代码行的行号。

表 4-20 预定义宏 （续）

名称	值	定义时间
__MODULE__	*mod*	包含 __FILE__ 值的文件名部分。
__NO_INLINE__	–	在 GNU 模式下指定 --no_inline 选项时。
__OPTIMISE_LEVEL	*num*	缺省情况下，始终设置为 2，除非使用 -O*num* 选项更改了优化级别。
__OPTIMISE_SPACE	–	指定 -Ospace 选项时。
__OPTIMISE_TIME	–	指定 -Otime 选项时。
__OPTIMIZE__	–	在 GNU 模式下指定 -O1、-O2 或 -O3 时。
__OPTIMIZE_SIZE__	–	在 GNU 模式下指定 -Ospace 时。
__PLACEMENT_DELETE	–	在 C++ 模式下指定启用位置删除 （即，在构造函数抛出异常时调用的与位置运算符 **new** 对应的运算符 **delete**）。这仅适用于使用异常的情况。
__PTRDIFF_TYPE__	–	在 GNU 模式下。它为 ptrdiff_t **typedef** 定义正确的基础类型。
__RTTI	–	在 C++ 模式下启用 RTTI 时。
__sizeof_int	4	用于 sizeof(int)，但在预处理程序表达式中可用。
__sizeof_long	4	用于 sizeof(long)，但在预处理程序表达式中可用。
__sizeof_ptr	4	用于 sizeof(void *)，但在预处理程序表达式中可用。
__SIZE_TYPE__	–	在 GNU 模式下。它为 size_t **typedef** 定义正确的基础类型。
__SOFTFP__	–	如果编译为使用软件浮点调用标准和库。为 ARM 或 Thumb 指定 --fpu=softvfp 选项或者为 Thumb 指定 --fpu=softvfp+vfpv2 时设置。
__STDC__	–	在所有编译器模式下。
__STDC_VERSION__	–	标准版本信息。
__STRICT_ANSI__	–	指定 --strict 选项时。
__SUPPORT_SNAN__	–	指定 --fpmode=ieee_fixed 或 --fpmode=ieee_full 时支持信号 NaN。
__TARGET_ARCH_ARM	*num*	目标 CPU 的 ARM 基本体系结构编号，与编译器为 ARM 还是为 Thumb 进行编译无关。若要了解与 ARM 体系结构版本有关的 __TARGET_ARCH_ARM 可能值，请参阅第4-121 页的表 4-21。

表 4-20 预定义宏 （续）

名称	值	定义时间
__TARGET_ARCH_THUMB	*num*	目标 CPU 的 Thumb 基本体系结构编号，与编译器为 ARM 还是为 Thumb 进行编译无关。如果目标不支持 Thumb，则将该值定义为零。若要了解与 ARM 体系结构版本有关的 __TARGET_ARCH_THUMB 可能值，请参阅第4-121 页的表 4-21。
__TARGET_ARCH_*XX*	–	*XX* 表示目标体系结构，它的值取决于目标体系结构。例如，如果指定编译器选项 --cpu=4T 或 --cpu=ARM7TDMI，则会定义 __TARGET_ARCH_4T。
__TARGET_CPU_*XX*	–	*XX* 表示目标 CPU。*XX* 值是从 --cpu 编译器选项中获取的，如果没有指定任何选项，则使用缺省值。例如，如果指定编译器选项 --cpu=ARM7TM，则会定义 __TARGET_CPU_ARM7TM，而不会定义以 __TARGET_CPU_ 开头的任何其他符号。 如果指定了目标体系结构，则会定义 __TARGET_CPU_generic。 如果处理器名称包含连字符 (-)，则会将其映射为下划线 (_)。例如，将 --cpu=ARM1136JF-S 映射为 __TARGET_CPU_ARM1136JF_S。
__TARGET_FEATURE_DOUBLEWORD	–	ARMv5T 和更高版本。
__TARGET_FEATURE_DSPMUL	–	如果可以使用 DSP 增强的乘法器，例如 ARMv5TE。
__TARGET_FEATURE_MULTIPLY	–	如果目标体系结构支持长整型乘法指令 MULL 和 MULAL。
__TARGET_FEATURE_DIVIDE	–	如果目标体系结构支持硬件除法指令 （即， ARMv7-M 或 ARMv7-R）。
__TARGET_FEATURE_MULTIPROCESSING	–	指定以下任一选项时： • --cpu=Cortex-A9 • --cpu=Cortex-A9.no_neon • --cpu=Cortex-A9.no_neon.no_vfp
__TARGET_FEATURE_THUMB	–	如果目标体系结构支持 Thumb、 ARMv4T 或更高版本。

表 4-20 预定义宏 （续）

名称	值	定义时间
__TARGET_FPU_*xx*	-	设置以下宏之一以指示使用 FPU 的情形： • __TARGET_FPU_NONE • __TARGET_FPU_VFP • __TARGET_FPU_SOFTVFP 此外，如果使用以下 --fpu 选项进行编译，则相应的目标名称设置为： • --fpu=softvfp+vfpv2, __TARGET_FPU_SOFTVFP_VFPV2 • --fpu=softvfp+vfpv3, __TARGET_FPU_SOFTVFP_VFPV3 • --fpu=softvfp+vfpv3_fp16, __TARGET_FPU_SOFTVFP_VFPV3_FP16 • --fpu=softvfp+vfpv3_d16, __TARGET_FPU_SOFTVFP_VFPV3_D16 • --fpu=softvfp+vfpv3_d16_fp16, __TARGET_FPU_SOFTVFP_VFPV3_D16_FP16 • --fpu=vfpv2, __TARGET_FPU_VFPV2 • --fpu=vfpv3, __TARGET_FPU_VFPV3 • --fpu=vfpv3_fp16, __TARGET_FPU_VFPV3_FP16 • --fpu=vfpv3_d16, __TARGET_FPU_VFPV3_D16 • --fpu=vfpv3_d16_fp16, __TARGET_FPU_VFPV3_D16_FP16 有关详细信息，请参阅第2-61 页的*--fpu=name*。
__TARGET_PROFILE_A		指定 --cpu=7-A 选项时。
__TARGET_PROFILE_R		指定 --cpu=7-R 选项时。
__TARGET_PROFILE_M		指定以下任一选项时： • --cpu=6-M • --cpu=6S-M • --cpu=7-M

表 4-20 预定义宏 （续）

名称	值	定义时间
__thumb__	–	编译器处于 Thumb 模式时。即，在命令行指定了 --thumb 选项或在源代码中指定了 #pragma thumb。 **注意** • 即使为 Thumb 进行编译，编译器也可能会生成某些 ARM 代码。 • 在使用 #pragma thumb 或 #pragma arm 时，将会定义或不定义 __thumb 或 __thumb__，但在作为 ARM 代码生成 Thumb 函数的情况下，它们不会由于其他原因 （例如，将函数指定为 __irq）而发生改变。
__TIME__	*time*	始终定义。
__UINTMAX_TYPE__	–	在 GNU 模式下。它为 uintmax_t **typedef** 定义正确的基础类型。
__USER_LABEL_PREFIX__		在 GNU 模式下。它定义空字符串。此宏由一些 Linux 头文件使用。
__VERSION__	*ver*	指定 --gnu 选项时。它是一个字符串，用于显示所使用的 GNU 模式的当前版本。
_WCHAR_T	–	在 C++ 模式下，将 **wchar_t** 指定为关键字。
__WCHAR_TYPE__	–	在 GNU 模式下。它为 wchar_t **typedef** 定义正确的基础类型。
__WCHAR_UNSIGNED__	–	在 GNU 模式下指定 --cpp 选项时。仅当 **wchar_t** 为无符号类型时，才会对其进行定义。
__WINT_TYPE__	–	在 GNU 模式下。它为 wint_t **typedef** 定义正确的基础类型。

表 4-21 显示了 __TARGET_ARCH_THUMB 的可能值 （请参阅第4-115 页的表 4-20），以及这些值与 ARM 体系结构版本之间的关系。

表 4-21 Thumb 体系结构版本与 ARM 体系结构版本的关系

ARM 体系结构	__TARGET_ARCH_ARM	__TARGET_ARCH_THUMB
v4	4	0
v4T	4	1
v5T、 v5TE、 v5TEJ	5	2
v6、 v6K、 v6Z	6	3
v6T2	6	4
v6-M、 v6S-M	0	3
v7-A、 v7-R	7	4
v7-M	0	4

4.10.2 函数名称

表 4-22 列出了 ARM 编译器为 C 和 C++ 支持的内置变量。

表 4-22 内置变量

名称	值
__FUNCTION__	保留函数名称，它与源代码中显示的名称相同。 __FUNCTION__ 是一个常数字符串文字。无法通过使用预处理程序，将内容与其他文本合并在一起以组成新的标记。
__PRETTY_FUNCTION__	保留函数名称，它基本上是以语言特定的方式输出的。 __PRETTY_FUNCTION__ 是一个常数字符串文字。无法通过使用预处理程序，将内容与其他文本合并在一起以组成新的标记。

第 5 章
C 和 C++ 实现细节

本章介绍 ARM 编译器的语言实现细节。本章内容如下：

5.1 C 和 C++ 实现细节

本节介绍适用于 C 和 C++ 的语言实现细节。

5.1.1 字符集和标识符

以下内容适用于编译器所需的字符集和标识符：

- 在所有内部和外部标识符中，大写和小写字符是不同的。除非指定 `--strict` 编译器选项，否则，标识符还可能包含美元 (`$`) 字符。在使用 `--strict` 选项的情况下，要允许标识符中包含美元符号，还应使用 `--dollar` 命令行选项。
- 通过调用 `setlocale(LC_CTYPE, "ISO8859-1")`，可使 `isupper()` 和 `islower()` 函数按预期方式使用完整 8 位 Latin-1 字符集，而不是使用 7 位 ASCII 子集。必须在链接时选择语言环境。
- 源文件会根据当前所选的语言环境进行编译。如果源文件包含非 ASCII 字符，则可能需要使用 `--locale` 命令行选项选择其他语言环境。有关详细信息，请参阅 《编译器用户指南》中第2-2 页的*调用 ARM 编译器*。
- ARM 编译器支持多字节字符集，例如 Unicode。
- 源代码字符集的其他属性因主机而异。

执行字符集的属性因目标而异。 ARM C 和 C++ 库支持 ISO 8859-1 （Latin-1 字母表）字符集，并产生以下结果：

- 执行字符集与源代码字符集完全相同。
- 执行字符集中的字符有 8 位。
- **`int`** 中有 4 个字符 （字节）。如果内存系统为：

  小端 字节的排列顺序是：最低有效位在最低地址，最高有效位在最高地址。

  大端 字节的排列顺序是：最低有效位在最高地址，最高有效位在最低地址。
- 在 C 中，所有字符常数的类型都为 **`int`**。在 C++ 中，包含一个字符的字符常数的类型为 **`char`**，包含多个字符的字符常数的类型为 **`int`**。整型值最多可以表示 4 个常数字符。常数中的最后一个字符占用整型值的最低顺序字节。前面最多三个字符放在更高顺序的字节。将使用 `NULL` (`\0`) 字符填充未用字节。

- 表 5-1 列出了包含单个字符或字符转义序列的所有整型字符常数，并用源代码和执行字符集表示它们。

表 5-1 字符转义码

转义序列	字符值	说明
\a	7	注意 （铃声）
\b	8	退格符
\t	9	水平制表符
\n	10	换行符
\v	11	垂直制表符
\f	12	换页符
\r	13	回车符
\xnn	0xnn	十六进制的 ASCII 码
\nnn	0nnn	八进制的 ASCII 码

- 字符串文字和字符常数中的源代码字符集字符等同地映射到执行字符集。
- 缺省情况下，**char** 类型的数据项没有符号。可以显式地将它们声明为 **signed char** 或 **unsigned char**：
  — 可以使用 --signed_chars 选项将 **char** 变为有符号的数据。
  — 可以使用 --unsigned_chars 选项将 **char** 变为无符号的数据。

  注意

  如果要混合已使用和未使用 --signed_chars 和 --unsigned_chars 选项编译的转换单元，并且它们使用相同的接口或数据结构，则必须格外小心。

  ARM ABI 将 **char** 定义为无符号字节，这是 RVCT 附带提供的 C++ 库使用的解释。
- 不使用任何语言环境将多字节字符转换为宽字符常数的相应宽字符。这与通常的实现无关。

5.1.2 基本数据类型

本节介绍了如何在 ARM C 和 C++ 中实现基本数据类型。

基本数据类型的大小和对齐

表 5-2 列出了基本数据类型的大小和自然对齐。

表 5-2 数据类型的大小和对齐

类型	大小 （位）	自然对齐 （字节）
`char`	8	1 （字节对齐）
`short`	16	2 （半字对齐）
`int`	32	4 （字对齐）
`long`	32	4 （字对齐）
`long long`	64	8 （双字对齐）
`float`	32	4 （字对齐）
`double`	64	8 （双字对齐）
`long double`	64	8 （双字对齐）
所有指针	32	4 （字对齐）
`bool` （仅适用于 C++）	8	1 （字节对齐）
`_Bool` （仅适用于 C[a]）	8	1 （字节对齐）
`wchar_t` （仅适用于 C++）	16	2 （半字对齐）

a. 可以使用 `stdbool.h` 在 C 中定义 `bool` 宏。

类型对齐因上下文而异：

- 局部变量通常保存在寄存器中，但在局部变量溢出到堆栈中时，它们始终是字对齐的。例如，溢出的局部变量 `char` 以 4 为边界对齐。
- 压缩类型的自然对齐边界为 1。

有关详细信息，请参阅第5-7 页的*结构、联合、枚举和位域*。

整数

整数以 2 的补码形式表示。在小端模式下，**long long** 的低字位于低位地址；在大端模式下，其低字位于高位地址。

浮点

浮点量以 IEEE 格式进行存储：

- **float** 值由 IEEE 单精度值表示
- **double** 和 **long double** 值由 IEEE 双精度值表示。

对于 **double** 和 **long double** 量：在大端模式下，包含符号、指数和尾数的最有效部分的字存储在低机器地址中；在小端模式下，则存储在高机器地址中。有关详细信息，请参阅第5-6 页的*浮点类型运算*。

数组和指针

以下说明适用于 C 和 C++ 中的所有指向对象的指针，而不适用于指向成员的指针：

- 相邻字节的地址相差 1。
- 宏 `NULL` 扩展为值 0。
- 整数和指针之间的类型转换不会改变表示形式。
- 对于指向函数的指针和指向数据的指针之间的类型转换，编译器将会发出警告。
- `size_t` 类型被定义为 `unsigned int`。
- `ptrdiff_t` 类型被定义为 `signed int`。

5.1.3 基本数据类型运算

ARM 编译器执行 ISO C99 和 ISO C++ 标准的相关部分中介绍的常见算术转换。以下几个小节介绍了与算术运算有关的其他内容。

另请参阅第B-7 页的*表达式求值*。

整型运算

以下说明适用于整型运算：

- 所有有符号的整型算法使用 2 的补码表示形式。
- 有符号整型的位运算遵循根据 2 的补码表示形式自然形成的规则。不会进行符号扩展。
- 有符号量的右移是算术移位。
- 对于 **int** 类型的值，
  - 没有定义在范围 0 到 127 以外的移位。
  - 超过 31 位的左移结果为零。
  - 无符号值或有符号正值超过 31 位的右移结果为零。有符号负值的移位结果为 –1。
- 对于 **long long** 类型的值，没有定义在范围 0 到 63 以外的移位。
- 按照 ISO C99 标准的规定，整数除法余数的符号与被除数相同。
- 如果整型值被截断为较短的有符号整型，可通过丢弃适当数量的最高有效位来获得结果。对于新类型来说，如果原始数是太大的正或负数，则无法保证结果的符号与原始数相同。
- 整型之间的转换不会产生异常。
- 整数溢出不会产生异常。
- 缺省情况下，整数被零除将返回零。

浮点类型运算

以下说明适用于浮点类型运算：

- 标准 IEEE 754 规则适用。
- 缺省情况下，舍入到最接近的可表示的值。
- 缺省情况下，禁用浮点异常。

另请参阅第2-59 页的*--fpmode=model*。

——注意——

适用于浮点处理的 IEEE 754 标准规定，对异常执行的缺省操作是继续运行而不进行捕获。可通过调整 fenv.h 中的函数和定义来修改浮点错误处理方式。有关详细信息，请参阅第2-58 页的*调整错误信号*、*错误处理和程序退出*。

指针减法

以下说明适用于 C 中的所有指针。它们也适用于 C++ 中除指向成员的指针之外的其他指针：

- 从一个指针中减去另一个指针时，它们的差是以下表达式的结果：
  `((int)a - (int)b) / (int)sizeof(type pointed to)`
- 如果指针指向的对象的对齐边界与其大小相同，这种对齐方式可保证除法的准确性。
- 如果指针指向的对象的对齐边界小于其大小 （如压缩类型和大多数 **struct**），则两个指针必须指向相同数组的元素。

5.1.4 结构、联合、枚举和位域

本节介绍了结构化数据类型 union、enum 和 struct 实现；还讨论了结构填充和位域实现。

有关详细信息，请参阅第3-19 页的*匿名类*、*结构和联合*。

联合

使用其他类型的成员访问 **union** 的成员时，可以从原始类型的表示形式预测产生的值。不会生成任何错误。

枚举

enum 类型的对象是使用包含 **enum** 范围的最小整型实现的。根据 **enum** 中的枚举器范围， **enum** 的存储类型为以下内容的前者：

- **unsigned char**，如果未使用 `--enum_is_int`
- **signed char**，如果未使用 `--enum_is_int`
- **unsigned short**，如果未使用 `--enum_is_int`
- **signed short**，如果未使用 `--enum_is_int`
- **signed int**

- **unsigned int**， C 使用 --strict 时除外
- **signed long long**， C 使用 --strict 时除外
- **unsigned long long**， C 使用 --strict 时除外

通过按这种方式实现 **enum**，可以减小数据大小。命令行选项 --enum_is_int 强制将 **enum** 的基础类型设置为至少与 **int** 一样宽。

有关详细信息，请参阅 *《ARM 体系结构的过程调用标准》*规范中的 C 语言映射说明。

注意

如果要混合已使用和未使用 --enum_is_int 选项编译的转换单元，并且它们使用相同的接口或数据结构，则必须格外小心。

处理超出范围的值

在严格 C 中，必须能够将枚举器值表示为 **int**，例如，它们必须在 -2147483648 到 +2147483647 范围内 （含这两个值）。在早期版本的 RVCT 中，超出范围的值将转换为 **int**，而不会发出警告 （除非指定了 --strict 选项）。

在 RVCT 2.2 版和更高版本中，枚举器值超出范围时，将会发出警告：

```
#66: enumeration value is out of "int" range
```

这些值的处理方式与 C++ 相同，即，将它们作为 **unsigned int**、**long long** 或 **unsigned long long** 进行处理。

要确保报告超出范围的警告，请使用以下命令将它们更改为错误：

```
armcc --diag_error=66 ...
```

结构

以下内容适用于：

- 所有 C 结构
- 所有不使用虚拟函数或基类的 C++ 结构和类。

结构对齐

非压缩结构的对齐边界是其任何字段所需的最大对齐边界。

字段对齐

结构的排列方式为：将第一个已命名组件放在最低地址中。字段按以下方式进行对齐：

- **char** 类型的字段与下一个可用字节对齐。
- **short** 类型的字段与下一个偶数地址的字节对齐。
- 在 RVCT 2.0 和更高版本中，**double** 和 **long long** 数据类型为 8 字节对齐。这样，便可有效地使用 ARMv5TE 和更高版本中的 LDRD 和 STRD 指令。
- 位域对齐取决于位域的声明方式。有关详细信息，请参阅第5-12 页的*压缩结构中的位域*。
- 所有其他类型按字边界对齐。

结构可以包含填充以确保正确对齐字段以及结构本身。图 5-1 是一个常规非压缩结构的示例。它填充了第 1、第 2 和第 3 个字节以确保正确对齐字段。它还填充了第 11 和第 12 个字节以确保正确对齐结构。sizeof() 函数返回结构大小（包括填充）。

```
struct {char c; int x; short s} ex1;
```

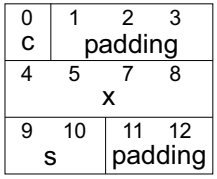

图 5-1 常规非压缩结构示例

根据结构的定义方式，编译器使用以下方式之一填充结构：

- 用零填充定义为 **static** 或 **extern** 的结构。
- 使用以前存储在堆栈或堆中的任何内容填充这些内存位置上的结构，例如，使用 malloc() 或 **auto** 定义的结构。不能使用 memcmp() 比较以这种方式定义的填充结构 （请参阅图 5-1）。

可以使用 --remarks 选项查看编译器在 **struct** 中插入填充时生成的消息。

C++ 中允许使用带有空初始值设定项的结构：

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

然而，如果使用 -cpp 和 --c90 选项编译 C 或 C++，则会生成错误。

压缩结构

压缩结构是一种结构对齐边界和内部字段对齐边界始终为 1 的结构。

可以使用 __packed 限定符压缩特定结构。或者，也可以使用 #pragma pack(*n*) 确保压缩任何包含未对齐数据的结构。无法使用命令行选项更改结构的缺省压缩方式。

位域

在非压缩结构中， ARM 编译器在*容器*中分配位域。容器是已声明类型的正确对齐的对象。

分配位域时，应确保指定的第一个字段占用字的最低地址位，具体取决于配置：

小端 最低地址为最低有效位。

大端 最低地址为最高有效位。

位域容器可以是任何整型。

——注意——

在严格 1990 ISO 标准 C 中，只允许将 **int**、**signed int** 和 **unsigned int** 类型用于位域。对于非 **int** 位域，编译器会显示错误。

没有使用 **signed** 或 **unsigned** 限定符声明的普通位域按 **unsigned** 处理。例如，int x:10 分配 10 位无符号整数。

位域将分配给第一个具有足够未分配位数的正确类型的容器，例如：

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

第一个声明创建一个整数容器，并为 x 分配 10 位。在第二个声明中，编译器查找具有足够未分配位数的现有整数容器，并在与 x 相同的容器中分配 y。

位域完全包含在其容器中。如果无法将位域放入某个容器中，则会将其放在下一个相同类型的容器中。例如，如果为结构声明了其他位域， z 声明将溢出容器：

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

编译器填充第一个容器的剩余两位，并为 z 分配一个新的整数容器。

位域容器可以相互*重叠*，例如：

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

第一个声明创建一个整数容器，并为 x 分配 10 位。这 10 位占用该整数容器的第 1 个字节以及第 2 个字节的两位。在第二个声明中，编译器检查 **char** 类型的容器。由于没有适合的容器，因此，编译器分配一个正确对齐的新 **char** 容器。

由于 **char** 的自然对齐边界为 1，因此，编译器搜索包含足够未分配位数的第一个字节，以便完全包含该位域。在示例结构中，**int** 容器的第二个字节为 x 分配了两位，有六位没有分配。编译器分配一个 **char** 容器（从前一个 **int** 容器的第二个字节开始），跳过分配给 x 的前两位，然后为 y 分配两位。

如果将 y 声明为 `char y:8`，则编译器填充第二个字节，然后为第三个字节分配一个新的 **char** 容器，因为位域不能溢出其容器。图 5-2 显示了以下示例结构的位域分配情况：

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

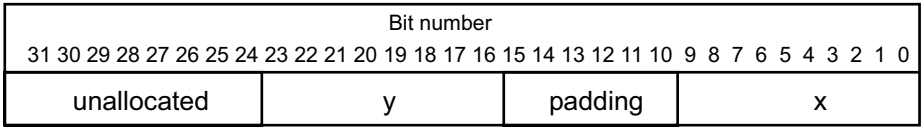

图 5-2 位域分配 1

注意

相同的基本规则适用于具有不同容器类型的位域声明。例如，要为示例结构添加 **int** 位域，请使用以下代码：

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

编译器分配一个 **int** 容器（该容器的起始位置与 `int x:10` 容器相同），并分配一个字节对齐的 **char** 和一个 5 位的位域，请参阅图 5-3。

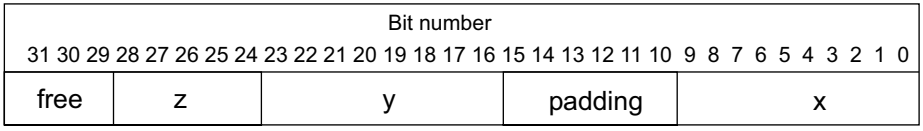

图 5-3 位域分配 2

通过声明零大小的未命名位域，可以显式地填充位域容器。如果容器不为空，则使用零大小的位域填充容器，直至末尾。后续位域声明开始一个新的空容器。

压缩结构中的位域

压缩结构中的位域容器的对齐边界为 1。因此，压缩结构中位域的最大位填充是 7 位。对于非压缩结构，最大填充是 `8*sizeof(container-type)-1` 位。

5.2 C++ 实现细节

本节介绍了 C++ 特定的语言实现细节。

5.2.1 使用 ::operator new 函数

根据 ISO C++ 标准，在内存分配失败时，`::operator new(std::size_t)` 将抛出异常，而不是发出信号。如果未捕获到异常，则会调用 `std::terminate()`。

编译器选项 `--force_new_nothrow` 将编译中的所有新调用转变为 `::operator new(std::size_t, std::nothrow_t&)` 或 `:operator new[](std::size_t, std::nothrow_t&)` 调用。但是，这并不影响库中的 `operator new` 调用，也不影响对任何类特定的 `operator new` 的调用。有关详细信息，请参阅第2-57 页的 *--force_new_nothrow, --no_force_new_nothrow*。

旧对象支持

在 RVCT 2.0 版中，当 `::operator new` 函数出现内存不足时，它将发出信号 **SIGOUTOFHEAP**，而不是引发 C++ 异常。请参阅 *《库和浮点支持指南》*中第2-95 页的*ISO C 库实现定义*。

在当前版本中，可以安装 `new_handler` 以发出信号，从而恢复 RVCT 2.0 版行为。

——注意——

不要依赖此行为的实现细节，因为它在将来版本中可能会发生变化。

5.2.2 试验数组

ADS 1.2 和 RVCT 1.2 版 C++ 编译器可以使用试验（即不完整）数组声明，例如 `int a[]`。使用 RVCT 2.x 或更高版本的编译器编译 C++ 时，不能使用试验数组。

5.2.3 C++ 函数中的旧式 C 参数

ADS 1.2 和 RVCT 1.2 版 C++ 编译器允许在 C++ 函数中使用旧式 C 参数。即，

```
void f(x) int x; { }
```

在 RVCT 2.x 或更高版本的编译器中，如果代码在函数中包含任何旧式参数，则必须使用 `--anachronisms` 编译器选项。如果编译器发现任何实例，则会发出警告。

5.2.4 过时功能

使用 --anachronisms 选项启用过时功能后，可以接受以下过时功能：

- 允许在函数声明中使用 **overload**。将接受并忽略重载。
- 可使用缺省初始化进行初始化的静态数据成员不需要定义。过时功能不适用于模板类的静态数据成员，因为必须始终定义这些成员。
- 可以在数组删除操作中指定数组中的元素个数。将忽略该值。
- 可以使用单个 operator++() 和 operator--() 函数重载前缀和后缀操作。
- 如果只有一个直接基类，则可以在基类初始值设定项中省略基类名称。
- 允许在构造函数和析构函数中分配 this 指针。
- 可以将边界函数指针 （即指向给定对象的成员函数的指针）转换为指向某个函数的指针。
- 嵌套的类名可以用作非嵌套的类名，但前提是尚未声明具有该名称的其他类。过时功能不适用于模板类。
- 可以通过其他类型的值初始化非 const 类型的引用。将创建一个临时变量，通过转换的初始值对其进行初始化，然后将引用设置为该临时变量。
- 可以通过非 const 类类型的右值或从该类类型派生类的值初始化对该类类型的引用。不需要使用额外的临时变量。
- 允许使用带有旧式参数声明的函数，该函数可以参与函数重载，就好像它是原型函数一样。进行兼容性检查时，不会对此类函数的参数类型应用缺省自变量升级，因此以下代码声明了两个名为 f 的函数的重载：

  ```
  int f(int);
  int f(x) char x; { return x; }
  ```

  注意

  在 C 中，此代码是合法的，但具有不同的含义。f 试验声明的后面是其定义。

5.2.5 模板实例化

ARM 编译器自动进行所有模板实例化，并确保链接后每个模板实体只保留一个定义。编译器通过在已命名公共节中发出模板实体来实现此功能。因此，链接器将删除所有重复的公共节 （即具有相同名称的公共节）。

——注意——

可以使用 --pending_instantiations 编译器选项限制给定模板的并发实例化次数。

有关详细信息，另请参阅第2-101 页的*--pending_instantiations=n*。

隐式包含

启用隐式包含后，编译器假定，如果它需要定义来实例化在 .h 文件中声明的模板实体，则可以隐式地包含相应的 .cc 文件以获取该定义的源代码。例如，如果在 xyz.h 文件中声明了模板实体 ABC::f，并且在编译中需要实例化 ABC::f，但是编译处理的源代码中没有出现 ABC::f 定义，编译器将检查 xyz.cc 文件是否存在。如果该文件存在，编译器将处理该文件，就像将该文件包含在主源文件末尾一样。

要查找给定模板实体的模板定义文件，编译器必须知道用于声明模板的文件的完整路径名，以及是否使用系统包含语法 （例如，#include <file.h>）包含了该文件。对于包含 #line 指令的预处理源代码，该信息不可用。因此，编译器不会尝试隐式包含含有 #line 指令的源代码。

编译器查找定义文件后缀 .cc 和 .CC。

可以使用命令行选项 --implicit_include 和 --no_implicit_include 打开或关闭隐式包含模式。

仅在正常文件编译期间执行隐式包含，即在未使用 -E 命令行选项时。

有关详细信息，请参阅第2-2 页的*命令行选项*。

5.2.6 命名空间

在模板实例化中进行名称查找时，必须在模板定义的上下文中查找某些名称，可以在模板实例化的上下文中查找其他名称。编译器实现两种不同的实例化查找算法：

- 标准规定的算法，这种算法称为相关名称查找。
- 在实现相关名称查找之前存在的算法。

相关名称查找是在 strict 模式下进行的，除非其他命令行选项显式地将其禁用，或者由配置标记或命令行选项启用相关名称处理。

相关名称查找处理

在进行相关名称查找时，编译器将实现标准中规定的实例化名称查找规则。该处理要求进行非类原型实例化。而这又需要使用标准要求的类型名称和模板关键字编写代码。

使用引用上下文进行查找

在不使用相关名称查找时，编译器使用与标准的两阶段查找规则近似的名称查找算法，但在某种程度上，它与现有代码和现有编译器更兼容。

如果将某个名称作为模板实例化一部分进行查找，但在实例化的局部上下文中找不到该名称，则在综合实例化上下文中进行查找。该综合实例化上下文包括模板定义上下文中的名称以及实例化上下文中的名称。例如：

```
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)
}
```

模板实例化中的名称查找在以下几个方面并不符合标准中的规则：

- 虽然只有模板定义上下文中的名称被视为不是函数的名称，但是查找并不限于在定义模板处可见的那些名称。
- 对于模板中的所有函数调用，将考虑引用模板的上下文中的函数。引用上下文中的函数只对相关函数调用可见。

与自变量相关的查找

启用与自变量相关的查找后，通过与自变量相关的查找显示的函数可能与通过正常查找显示的函数发生重载。标准要求即使在正常查找发现的名称为块 extern 声明时，也要进行这种重载。编译器可实现这种重载，但在缺省模式下，在正常查找发现块 extern 时禁止与自变量相关的查找。

这意味着，即使程序不使用命名空间，程序也可能具有不同的行为，具体取决于是否使用与自变量相关的查找对其进行编译。例如：

```
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++ 异常处理

RVCT 中完全支持 C++ 异常处理。但是，在缺省情况下，编译器不支持此功能。您必须使用 --exceptions 选项启用 C++ 异常处理。有关详细信息，请参阅第2-53 页的*--exceptions*, *--no_exceptions*。

注意

Rogue Wave 标准 C++ 库是在启用 C++ 异常的情况下提供的。

您可以对异常表生成进行有限的控制。

运行时的函数展开

缺省情况下，可以在运行时展开使用 --exceptions 编译的函数。有关详细信息，请参阅第2-53 页的*--exceptions*, *--no_exceptions*。*函数展开*包括破坏 C++ 自动变量并恢复在堆栈帧中保存的寄存器值。函数展开是通过发出描述要执行的操作的异常表实现的。

可以使用编译指示 #pragma exceptions_unwind 和 #pragma no_exceptions_unwind 为特定函数启用或禁用展开。有关详细信息，请参阅第4-55 页的*编译指示*。--exceptions_unwind 选项设置此编译指示的初始值。

为函数禁用函数展开操作产生以下结果：

- 在运行时无法通过该函数抛出异常，并且该抛出不会展开堆栈。如果抛出语言为 C++，则会调用 std::terminate。
- 可以使用很小的异常表表示形式来描述函数，这可帮助智能链接器进行表优化。
- 由于调用方和被调用方必须正确地进行交互，函数内联将受到限制。

因此，可以使用 #pragma no_exceptions_unwind，以一种不需要额外源代码装饰的方式强制禁止展开。

与此相反，在 C++ 中，空函数异常规范允许进行函数展开 （包括受保护的函数），然后根据 ISO C++ 标准调用 std::unexpected()。

5.2.8 外部内联函数

ISO C++ 标准要求每次使用内联函数时都要对其进行定义。为防止内联函数的多个外联副本之间发生冲突， C++ 编译器在公共节中发出外联 extern 函数。

外联的内联函数

在以下情况下，编译器以外联方式发出内联函数：

- 获取函数的地址，例如：

```
inline int g()
{
    return 1;
}
int (*fp)() = &g;
```

- 无法内联函数，例如，递归函数：

```
inline unsigned int fact(unsigned int n) {
    return n < 2 ? 1 : n * fact(n - 1);
}
```

- 编译器使用试探法判定最好不要内联该函数。这种试探法受 -Ospace 和 -Otime 的影响。如果使用 -Otime，编译器将内联更多的函数。通过使用 __forceinline 声明函数，可以忽略这种试探法。例如：

```
__forceinline int g()
{
    return 1;
}
```

有关详细信息，另请参阅第2-58 页的*--forceinline*。

附录 A
via 文件语法

本附录介绍所有 ARM 开发工具都接受的 via 文件的语法。它分为以下几节：

- 第A-2 页的*via 文件概述*
- 第A-3 页的*语法*

A.1 via 文件概述

via 文件是无格式文本文件，包含 ARM 开发工具的命令行参数和选项。可以通过所有 ARM 命令行工具使用 via 文件，也就是说，可以将 --via 命令行选项和以下命令结合使用，从命令行指定 via 文件：

- armcc
- armasm
- armlink
- fromelf
- armar。

有关详细信息，请参阅各个工具的文档。

注意

在通常情况下，可以使用 via 文件对工具指定任意命令行选项，包括 --via。这意味着可以在 via 文件内调用多重嵌套的 via 文件。

本节包括以下内容：

- *via 文件求值*

A.1.1 via 文件求值

调用支持 via 文件的工具后：

1. 使用从 via 文件中提取的自变量字序列，替换最初指定的 --via *via_file* 自变量，包括递归处理 via 文件中的所有嵌套 --via 命令。
2. 以相同的方式，按参数出现的顺序处理所有后续 --via *via_file* 参数。

也就是说，按照指定的顺序处理 via 文件，并且在完全处理每一个 via 文件（包括处理嵌套的 via 文件）后，再处理下一个 via 文件。

A.2 语法

via 文件必须符合以下语法规则：

- via 文件是包含字序列的文本文件。该文本文件中的每个字都被转换为自变量字符串，然后传递给工具。
- 这些字由空格或行末分隔，已分隔字符串除外。例如：

  `--c90 --strict` （两个字）

  `--c90--strict` （一个字）
- 行末当作空格进行处理。例如：

  ```
  --c90
  --strict
  ```

  等效于：

  `--c90 --strict`
- 括在双引号 (") 或单引号 (') 中的字符串按照单个字处理。在用双引号括起来的字内，单引号按照普通字符处理。在用单引号分隔的字内，双引号按照普通字符处理。

  双引号用于分隔包含空格的文件名或路径名。例如：

  `-I C:\My Project\includes`（三个字）`-I "C:\My Project\includes"`（两个字）

  单引号可用于分隔包含双引号的字。例如：

  `-DNAME='"RealView Compilation Tools"'` （一个字）
- 用括号括起来的字符按照单个字处理。例如：

  `--option(x, y, z)` （一个字）

  `--option (x, y, z)` （两个字）
- 在用双引号或单引号分隔的字符串内，可以使用反斜杠 (\) 字符对双引号、单引号和反斜杠字符进行转义。
- 如果分隔的字旁边紧靠着另一个字，则按照单个字处理。例如：

  `-I"C:\Project\includes"`

  按照单个字处理：

  `-IC:\Project\includes`

- 以分号 (;) 或井字 (#) 字符作为第一个非空格字符开头的行是注释行。如果分号或井字字符出现在行中的任何其他位置，则不将其看作注释行的开头。例如：

  ```
  -o objectname.axf      ;this is not a comment
  ```

  注释以行末或文件尾结束。注释不能跨多行，也不能是行中的一部分。

- 包含预处理程序选项 -Dsymbol="value" 的行必须用单引号分隔，可表示为 '-D*symbol*="*value*"' 或 -Dsymbol='"value"'。例如：

  ```
  -c -DFOO_VALUE='"FOO_VALUE"'
  ```

附录 B
标准 C 实现定义

本附录提供 ISO C 标准所规定的符合条件的 C 实现的信息。它分为以下几节：

- 第B-2 页的*实现定义*
- 第B-9 页的*被视为 ISO C 标准未定义的行为*

B.1 实现定义

ISO C 标准 (ISO/IEC 9899:1990 (E)) 的附录 G 整理了有关可移植性的信息。子节 G3 列出了每一执行必须记录的行为。

注意

本附录不重复介绍第 4 章 *编译器特有的功能*中的信息。本附录在适当之处提供了引用。

以下各小节对应于子节 G3 的相关各节。这些小节介绍了 ARM C 编译器和 C 库等方面的内容，这些内容不是由 ISO C 标准定义的，而是由执行定义的：

注意

对 `wctype.h` 和 `wchar.h` 头文件的支持不包括宽文件操作。

B.1.1 转换

编译器生成的诊断消息的格式如下：

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

其中， *severity* 是下列值之一：

[blank] 如果严重性为空，则这是一个备注，表示对 C 或 C++ 的常见 （但有时是非约定的）用法。缺省情况下不显示备注。可使用 --remarks 选项来显示备注消息。有关详细信息，请参阅第6-4 页的*控制诊断消息的输出*。继续编译。

Warning 标记代码中表示可能出现问题的异常情形。继续编译。

Error 指示导致编译停止的问题。例如，违反了 C 或 C++ 语言的语法或语义规则。

Internal fault

指示编译器的内部问题。请与您的供应商联系，并准备好如第x 页的*反馈*所列的信息。

其中：

error-code 是标识错误类型的编号。

explanation 是对错误的文字描述。

有关详细信息，请参阅 《编译器用户指南》中第 6 章 *诊断消息*。

B.1.2 环境

基于 ARM 体系结构的环境的命令行到 main() 中参数的映射是与执行相关的。通用的 ARM C 库支持以下内容：

- *main()*
- 第B-4 页的*交互设备*
- 第B-4 页的*重定向标准输入、输出和错误流*

main()

赋予 main() 的参数是该命令行中一些不包括输入/输出重定向的字，字与字之间用空格分隔，但包含在双引号中的空格除外。

注意

- 在 isspace() 结果为 true 的情况下，空格字符是任意字符。

- 双引号或双引号之内的反斜杠字符 \ 的前面必须有一个反斜杠字符。
- 输入/输出重定向在双引号之内不会被识别。

交互设备

在 ARM C 库的非主机执行中，术语*交互设备*可能无意义。通用的 ARM C 库支持两个都称为 :tt 的一对设备，它们用于处理键盘输入和 VDU 屏幕输出。在通用执行中：

- 除非输入/输出重定向已发生，否则在连接 :tt 的任何流中都不进行缓冲处理
- 如果输入/输出已重定向到 :tt 以外的地方，则使用整个文件缓冲处理，但如果 stdout 和 stderr 都被重定向到同一个文件，则使用行缓冲处理。

重定向标准输入、输出和错误流

使用通用的 ARM C 库，可以在运行时重定向标准输入、输出和错误流。例如，如果 mycopy 是一个运行在主机调试器上的程序，该程序将标准输入复制到标准输出，则使用以下行运行该程序：

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

并按如下方式重定向文件：

stdin　　标准输入流重定向至 infile。

stdout　　标准输出流重定向至 outfile。

stderr　　标准错误流重定向至 errfile。

允许的重定向为：

0< *filename*　从 *filename* 读取 stdin。

< *filename*　从 *filename* 读取 stdin。

1> *filename*　将 stdout 写入 *filename*。

> *filename*　将 stdout 写入 *filename*。

2> *filename*　将 stderr 写入 *filename*。

2>&1　　将 stderr 写入与 stdout 相同的位置。

`>& file` 将 stdout 和 stderr 都写入 *filename*。

`>> filename` 将 stdout 附加至 *filename*。

`>>& filename` 将 stdout 和 stderr 都附加至 *filename*。

要重定向目标上的 stdin、stdout 和 stderr，必须进行以下定义：

```
#pragma import(_main_redirection)
```

仅在以下两种情况下进行文件重定向：

- 调用操作系统支持文件重定向
- 程序读写字符而并不替换 C 库函数 fputc() 和 fgetc()。

B.1.3 标识符

有关详细信息，请参阅第5-2 页的*字符集和标识符*。

B.1.4 字符

有关详细信息，请参阅第5-2 页的*字符集和标识符*。

B.1.5 整数

有关详细信息，请参阅第5-5 页的*整数*。

B.1.6 浮点数

有关详细信息，请参阅第5-5 页的*浮点*。

B.1.7 数组和指针

有关详细信息，请参阅第5-5 页的*数组和指针*。

B.1.8 寄存器

使用 ARM 编译器，您可以声明任意数目的具有存储类 **register** 的局部对象。

B.1.9 结构、联合、枚举和位域

ISO/IEC C 标准要求为结构化数据类型记录以下执行的详细信息：

- 使用其他类型的成员访问联合成员时的结果
- 结构成员的填充和对齐
- 普通 **int** 位域是按照 **signed int** 位域处理还是按照 **unsigned int** 位域处理
- 单元内位域的分配顺序
- 位域是否可以跨越存储单元的边界
- 选定为表示枚举类型值的整数类型。

有关详细信息，请参阅第 5 章 *C 和 C++ 实现细节*。

联合

有关信息，请参阅第5-7 页的*联合*。

枚举

有关信息，请参阅第5-7 页的*枚举*。

结构的填充和对齐

有关信息，请参阅第5-8 页的*结构*。

位域

有关信息，请参阅第5-10 页的*位域*。

B.1.10 限定符

具有 volatile 限定类型的对象作为字、半字或字节进行访问，具体访问方式由该对象的大小和对齐情况确定。对于大于一个字的 volatile 对象，对该对象的各部分的访问顺序是未定义的。对 volatile 位域的更新通常需要进行读取-修改-写入。对对齐的字、半字和字节类型的访问是原子访问。其他 volatile 访问不一定是原子访问。

否则，对于 volatile 限定对象的读取和写入根据源代码的直接指示发生，并按源代码指示的顺序执行。

B.1.11 表达式求值

编译器可以对仅涉及同等优先级关联运算符和交换运算符的表达式（甚至是含有括号的表达式）进行重新排序。例如，如果 a、b 和 c 是整数表达式，则 a + (b + c) 可以按照 (a + b) + c 进行求值。

在序列点之间，编译器可以按照任意顺序对表达式求值，而不考虑括号。因此，在序列点之间的表达式无论以何种顺序出现，都有可能出现副作用。

编译器可以按照任意顺序对函数参数求值。

相关标准未规定的有关求值顺序的所有方面，可能因以下情况而有所不同：

- 编译时的优化级别
- 所使用的编译器的版本。

B.1.12 预处理指令

ISO 标准 C 头文件可以按照标准中所描述的方式引用，例如 `#include <stdio.h>`。

支持可包括的源文件的带括号名称。编译器接受主机文件名或 UNIX 文件名。对于非 UNIX 主机上的 UNIX 文件名，编译器会尝试将该文件名翻译为等价的本地文件名。

第4-55 页的*编译指示中显示了可识别的 `#pragma` 指令*。

B.1.13 库函数

《库和浮点支持指南》中第1-2 页的*关于运行时库*中列出了 ISO C 库变体。

每一 C 库的准确性质对特定实现来说是唯一的。通用的 ARM C 库具有或支持以下功能：

- 宏 NULL 扩展为整型常数 0。
- 如果程序重新定义保留的外部识别符 （例如 printf），则在该程序与标准库链接时可能会出错。如果该程序未与标准库链接，则检测不到错误。
- __aeabi_assert() 函数输出有关 stderr 失败诊断的详细信息，然后调用 abort() 函数：

  *** assertion failed: *expression*, file *name*, line *number*

  注意

  assert 宏的行为取决于最近出现的 #include <assert.h> 的运行条件。有关详细信息，请参阅 《库和浮点支持指南》中第2-36 页的*从程序中退出*。

有关数学函数、宏、语言环境、信号和输入/输出的实现细节，请参阅 《库和浮点支持指南》中的第 2 章 *C 和 C++ 库*。

B.2 被视为 ISO C 标准未定义的行为

以下行为被视为 ISO C 标准未定义的行为：

- 在字符和字符串转义中，如果跟随在 \ 后面的字符没有特殊意义，则转义值是字符本身。例如，如果使用 \s，则会生成一个警告，因为它与 s 相同。
- 在缺省情况下，没有命名字段但至少有一个未命名字段的 **struct** 是可接受的，但在严格的 1990 ISO 标准 C 中，则会生成一个错误。

附录 C
标准 C++ 实现定义

在编译 C++ 时，ARM 编译器支持 ISO/IEC C++ 标准中介绍的大多数语言功能。本附录列出在标准中定义的 C++ 语言功能，并指明 ARM C++ 是否支持某一语言功能。它分为以下几节：

- 第C-2 页的*整型转换*
- 第C-3 页的*调用纯虚函数*
- 第C-4 页的*主要的语言支持特性*
- 第C-5 页的*标准 C++ 库实现定义*

注意

本附录不重复介绍属于标准 C 实现的信息。请参阅附录 B *标准 C 实现定义*。

在 ISO C 模式下编译 C++ 时，ARM 编译器与 ARM C 编译器相同。在涉及 C 或 C++ 特有的实现特性的位置，本文将予以指明。有关标准 C++ 扩展的信息，请参阅：

- 第3-14 页的*标准 C++ 语言扩展*
- 第3-6 页的*C++ 和 C90 中提供的 C99 语言功能*
- 第3-18 页的*标准 C 和标准 C++ 语言扩展*

C.1 整型转换

在整型转换过程中，如果目标类型有符号，在该值可按照目标类型和位域宽度表示时，该值保持不变。否则，该值将被截断以适合目标类型的大小。

注意

本节内容与 ISO/IEC 标准的 4.7 节 “整型转换”相关。

C.2 调用纯虚函数

调用纯虚函数是非法的。如果代码调用纯虚函数，则编译器将包含对库函数 `__cxa_pure_virtual` 的调用。

`__cxa_pure_virtual` 发出信号 **SIGPVFN**。缺省的信号处理程序输出错误消息，然后退出。有关详细信息，请参阅 *《库和浮点支持指南》*中第2-61 页的 *__default_signal_handler()*。

C.3 主要的语言支持特性

表 C-1 列出了此版本 ARM C++ 支持的主要语言功能。

表 C-1 主要的语言功能支持

主要特性	ISO/IEC 标准部分	支持
内核语言	1 至 13	是。
模板	14	是，但导出模板除外。
异常	15	是。
库	17 到 27	请参阅第C-5 页的*标准 C++ 库实现定义*和*《库和浮点支持指南》*。

C.4 标准 C++ 库实现定义

2.02.03 版 Rogue Wave 库提供了标准中定义的库子集，与 1999 ISO C 标准稍有不同。有关实现定义的信息，请参阅 *《库和浮点支持指南》*中第2-101 页的*标准 C++ 库实现定义*。

库可以与用户定义的函数一起使用，生成与目标相关的应用程序。请参阅 *《库和浮点支持指南》*中第1-2 页的*关于运行时库*。

附录 D
C 和 C++ 编译器实现限制

本附录列出了使用 ARM 编译器编译 C 和 C++ 代码时的实现限制。本附录分为以下几节：

D.1 C++ ISO/IEC 标准限制

ISO/IEC C++ 标准推荐了符合标准的编译器必须接受的最低限制。在编译器之间移植应用程序时，一定要注意这些限制。表 D-1 简要说明了这些限制。

在该表中，memory 的限制指示 ARM 编译器没有施加任何限制，可用内存施加的限制除外。

表 D-1 实现限制

说明	推荐	ARM
复合语句、迭代控制结构和选择控制结构的嵌套层数。	256	内存
条件包含的嵌套层数。	256	内存
修改声明中的算法、结构、联合或不完整类型的指针、数组和函数声明符个数 （任意组合）。	256	内存
完整表达式中的括号表达式的嵌套层数。	256	内存
内部标识符或宏名中的初始字符数。	1024	内存
外部标识符中的初始字符数。	1024	内存
一个转换单元中的外部标识符数。	65536	内存
在一个块中声明的具有块范围的标识符数。	1024	内存
在一个转换单元中同时定义的宏标识符数。	65536	内存
一个函数声明中的参数个数。	256	内存
一个函数调用中的自变量个数。	256	内存
一个宏定义中的参数个数。	256	内存
一个宏调用中的自变量个数。	256	内存
一行逻辑源代码中的字符数。	65536	内存
字符串文字或连接后生成的宽字符串文字中的字符数。	65536	内存
C 或 C++ 对象 （包括数组）的大小。	262144	4294967296
#include 文件的嵌套层数。	256	内存
switch 语句的 case 标签数，不包括任何嵌套 **switch** 语句的 case 标签。	16384	内存
单个类、结构或联合中的数据成员数。	16384	内存

表 D-1 实现限制 （续）

说明	推荐	ARM
单个枚举中的枚举常数个数。	4096	内存
单个 **struct** 声明列表中的嵌套类、结构或联合定义层数。	256	内存
`atexit()` 注册的函数个数。	32	33
直接和间接基类数。	16384	内存
单个类的直接基类数。	1024	内存
在单个类中声明的成员数。	4096	内存
类中的最终覆盖虚拟函数个数 （包括可访问和无法访问的函数）。	16384	内存
类的直接和间接虚拟基址数。	1024	内存
类的静态成员数。	1024	内存
类的友元声明数。	4096	内存
类中的访问控制声明数。	4096	内存
构造函数定义中的成员初始值设定项数。	6144	内存
一个标识符的范围限定数。	256	内存
嵌套的外部说明数。	1024	内存
模板声明中的模板自变量个数。	1024	内存
递归嵌套的模板实例化次数。	17	内存
每个测试块的处理程序数。	256	内存
单个函数声明中的抛出说明数。	256	内存

D.2 整数限制

表 D-2 给出了 ARM C 和 C++ 中的整数范围。该表的 Endpoint 列给出了范围端点的数值。Hex value 列给出了位模式 （十六进制），即 ARM 编译器对此值的解释。这些常数是在 limits.h 包含文件中定义的。

输入常数时，应注意选择大小和符号。在十进制以及十六进制/八进制中，对常数的解释是不同的。有关详细信息，请参阅相应 C 或 C++ 标准或任何推荐的 C 和 C++ 教科书，如第 viii 页的*更多参考出版物*中所述。

表 D-2 整数范围

常数	含义	值	十六进制值
CHAR_MAX	**char** 的最大值	255	0xFF
CHAR_MIN	**char** 的最小值	0	0x00
SCHAR_MAX	**signed char** 的最大值	127	0x7F
SCHAR_MIN	**signed char** 的最小值	- 128	0x80
UCHAR_MAX	**unsigned char** 的最大值	255	0xFF
SHRT_MAX	**short** 的最大值	32767	0x7FFF
SHRT_MIN	**short** 的最小值	- 32768	0x8000
USHRT_MAX	**unsigned short** 的最大值	65535	0xFFFF
INT_MAX	**int** 的最大值	2147483647	0x7FFFFFFF
INT_MIN	**int** 的最小值	- 2147483648	0x80000000
LONG_MAX	**long** 的最大值	2147483647	0x7FFFFFFF
LONG_MIN	**long** 的最小值	- 2147483648	0x80000000
ULONG_MAX	**unsigned long** 的最大值	4294967295	0xFFFFFFFF
LLONG_MAX	**long long** 的最大值	9.2E+18	0x7FFFFFFF FFFFFFFF
LLONG_MIN	**long long** 的最小值	- 9.2E+18	0x80000000 00000000
ULLONG_MAX	**unsigned long long** 的最大值	1.8E+19	0xFFFFFFFF FFFFFFFF

D.3 浮点数限制

本节介绍了浮点数的特征。

表 D-3 给出了浮点数的特征、范围和限制。这些常数是在 float.h 包含文件中定义的。

表 D-3 浮点限制

常数	含义	值
FLT_MAX	**float** 的最大值	3.40282347e+38F
FLT_MIN	**float** 的最小标准化正浮点数值	1.175494351e – 38F
DBL_MAX	**double** 的最大值	1.79769313486231571e+308
DBL_MIN	**double** 的最小标准化正浮点数值	2.22507385850720138e – 308
LDBL_MAX	**long double** 的最大值	1.79769313486231571e+308
LDBL_MIN	**long double** 的最小标准化正浮点数值	2.22507385850720138e – 308
FLT_MAX_EXP	**float** 类型以 2 为底的指数的最大值	128
FLT_MIN_EXP	**float** 类型以 2 为底的指数的最小值	– 125
DBL_MAX_EXP	**double** 类型以 2 为底的指数的最大值	1024
DBL_MIN_EXP	**double** 类型以 2 为底的指数的最小值	– 1021
LDBL_MAX_EXP	**long double** 类型以 2 为底的指数的最大值	1024
LDBL_MIN_EXP	**long double** 类型以 2 为底的指数的最小值	– 1021
FLT_MAX_10_EXP	**float** 类型以 10 为底的指数的最大值	38
FLT_MIN_10_EXP	**float** 类型以 10 为底的指数的最小值	– 37
DBL_MAX_10_EXP	**double** 类型以 10 为底的指数的最大值	308
DBL_MIN_10_EXP	**double** 类型以 10 为底的指数的最小值	– 307
LDBL_MAX_10_EXP	**long double** 类型以 10 为底的指数的最大值	308
LDBL_MIN_10_EXP	**long double** 类型以 10 为底的指数的最小值	– 307

表 D-4 介绍了浮点数的其他特征。这些常数也是在 float.h 包含文件中定义的。

表 D-4 其他浮点特征

常数	含义	值
FLT_RADIX	ARM 浮点数表示法的底 （基数）	2
FLT_ROUNDS	浮点数的舍入模式	（最接近的） 1
FLT_DIG	**float** 精度的十进制位数	6
DBL_DIG	**double** 精度的十进制位数	15
LDBL_DIG	**long double** 精度的十进制位数	15
FLT_MANT_DIG	**float** 精度的二进制位数	24
DBL_MANT_DIG	**double** 精度的二进制位数	53
LDBL_MANT_DIG	**long double** 类型的精度的二进制位数	53
FLT_EPSILON	对于 **float** 类型， 1.0 + x != 1.0 中 x 的最小正值	1.19209290e – 7F
DBL_EPSILON	对于 **double** 类型， 1.0 + x != 1.0 中 x 的最小正值	2.2204460492503131e – 16
LDBL_EPSILON	对于 **long double** 类型， 1.0 + x != 1.0 中 x 的最小正值	2.2204460492503131e – 16L

注意

- 将某个浮点数转换为更短的浮点数时，它将舍入到最接近的可表示数。
- 浮点算法符合 IEEE 754。

附录 E
使用 NEON 支持

本附录介绍此版本的 *RealView 编译工具* (RVCT) 对 NEON 内在函数的支持。

它分为以下几节：

E.1 简介

RVCT 提供在 ARM 和 Thumb 状态下为 Cortex-A8 处理器生成 NEON 代码的内在函数。 NEON 内在函数在头文件 `arm_neon.h` 中定义。头文件既定义内在函数，也定义一组向量类型。

ARMv7 之前的体系结构不支持 NEON 内在函数。构建较早版本的体系结构时，或者构建不包括 NEON 的 ARMv7 体系结构配置文件时，编译器将 NEON 内在函数视为普通函数调用。这会导致链接时出现错误。

E.2 向量数据类型

定义了以下类型来表示向量。 NEON 向量数据类型是根据以下模式命名的：

```
<type><size>x<number of lanes>_t
```

例如，int16x4_t 是一个包含四条向量线的向量，每条向量线包含一个有符号 16 位整数。表 E-1 列出了向量数据类型。

表 E-1 向量数据类型

int8x8_t	int8x16_t
int16x4_t	int16x8_t
int32x2_t	int32x4_t
int64x1_t	int64x2_t
uint8x8_t	uint8x16_t
uint16x4_t	uint16x8_t
uint32x2_t	uint32x4_t
uint64x1_t	uint64x2_t
float16x4_t	float16x8_t
float32x2_t	float32x4_t
poly8x8_t	poly8x16_t
poly16x4_t	poly16x8_t

某些内在函数使用以下格式的向量类型数组：

```
<type><size>x<number of lanes>x<length of array>_t
```

这些类型被视为包含名为 val 的单个元素的普通 C 结构。

以下是一个结构定义示例：

```
struct int16x4x2_t
{
    int16x4_t val[2];
};
```

为长度为 2 至 4 的数组定义了数组类型，其向量类型为表 E-1 列出的任何一种。

E.3 内在函数

本节介绍的内在函数紧密映射至 NEON 指令。每节开头是函数原型列表，并包含指定等效汇编器指令的注释。编译器会选择一条具有所需语义的指令，但不保证编译器会生成列出的指令。

内在函数使用类似于 NEON 统一汇编器语法的命名方案。即，每个内在函数的格式如下：

<opname><flags>_<type>

另外提供 q 标记来指定内在函数对 128 位向量进行运算。

例如：

- vmul_s16，表示两个有符号 16 位值的向量相乘。

  这编译为 VMUL.I16 d2, d0, d1。

- vaddl_u8，是指两个包含无符号 8 位值的 64 位向量按长型相加，结果为无符号 16 位值的 128 位向量。

  这编译为 VADDL.U8 q1, d0, d1。

注意

本手册中的内在函数原型使用以下类型注释：

__const(*n*)　参数 *n* 必须是编译时常数

__constrange(*min*, *max*)

参数必须是编译时常数，范围为 *min* 至 *max*

__transfersize(*n*)

内在函数从此指针加载 *n* 个字节。

注意

使用 __fp16 的 NEON 内在函数原型仅可用于具有 NEON 半精度 VFP 扩展的目标。若要启用 __fp16，请使用 --fp16_format 命令行选项。请参阅第2-59 页的*--fp16_format=format*。

E.3.1 加法

以下内在函数对向量进行加法运算。结果中的每条向量线都是对每个操作数向量中的相应向量线执行加法运算的结果。执行的运算如下：

- *向量加法：vadd -> Vr[i]:=Va[i]+Vb[i]*
- *向量长型加法：vadd -> Vr[i]:=Va[i]+Vb[i]*
- 第E-6 页的*向量宽型加法：vadd -> Vr[i]:=Va[i]+Vb[i]*
- 第E-6 页的*向量半加：vhadd -> Vr[i]:=(Va[i]+Vb[i])>>1*
- 第E-6 页的*向量舍入半加：vrhadd -> Vr[i]:=(Va[i]+Vb[i]+1)>>1*
- 第E-7 页的*向量饱和加法：vqadd -> Vr[i]:=sat<size>(Va[i]+Vb[i])*
- 第E-7 页的*高位半部分向量加法：- > Vr[i]:=Va[i]+Vb[i]*
- 第E-7 页的*高位半部分向量舍入加法*

向量加法：vadd -> Vr[i]:=Va[i]+Vb[i]

Vr、 Va、 Vb 具有相等的向量线大小。

```
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
```

向量长型加法：vadd -> Vr[i]:=Va[i]+Vb[i]

Va、 Vb 具有相等的向量线大小，结果为向量线宽度变成两倍的 128 位向量。

```
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
```

向量宽型加法：vadd -> 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
```

向量半加：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
```

向量舍入半加：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
```

向量饱和加法：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
```

高位半部分向量加法：-> 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
```

高位半部分向量舍入加法

```
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 乘法

以下内在函数提供包含乘法的运算。

向量乘法：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
```

向量乘加：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
```

向量长型乘加：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
```

向量乘减：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
```

向量长型乘减

```
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
```

向量高位饱和加倍乘法

```
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
```

向量高位饱和舍入加倍乘法

```
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
```

向量长型饱和加倍乘加

```
int32x4_t vqdmlal_s16(int32x4_t a, int16x4_t b, int16x4_t c); // VQDMLAL.S16 q0,d0,d0
int64x2_t vqdmlal_s32(int64x2_t a, int32x2_t b, int32x2_t c); // VQDMLAL.S32 q0,d0,d0
```

向量长型饱和加倍乘减

```
int32x4_t vqdmlsl_s16(int32x4_t a, int16x4_t b, int16x4_t c); // VQDMLSL.S16 q0,d0,d0
int64x2_t vqdmlsl_s32(int64x2_t a, int32x2_t b, int32x2_t c); // VQDMLSL.S32 q0,d0,d0
```

向量长型乘法

```
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
```

向量长型饱和加倍乘法

```
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 减法

以下内在函数提供包含减法的运算。

向量减法

```
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
```

向量长型减法：vsub -> 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
```

向量宽型减法：vsub -> 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
```

向量饱和减法

```
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
```

向量半减

```
int8x8_t   vhsub_s8(int8x8_t a, int8x8_t b);       // VHSUB.S8 d0,d0,d0
int16x4_t  vhsub_s16(int16x4_t a, int16x4_t b);    // VHSUB.S16 d0,d0,d0
int32x2_t  vhsub_s32(int32x2_t a, int32x2_t b);    // VHSUB.S32 d0,d0,d0
uint8x8_t  vhsub_u8(uint8x8_t a, uint8x8_t b);     // VHSUB.U8 d0,d0,d0
uint16x4_t vhsub_u16(uint16x4_t a, uint16x4_t b);  // VHSUB.U16 d0,d0,d0
uint32x2_t vhsub_u32(uint32x2_t a, uint32x2_t b);  // VHSUB.U32 d0,d0,d0
int8x16_t  vhsubq_s8(int8x16_t a, int8x16_t b);    // VHSUB.S8 q0,q0,q0
int16x8_t  vhsubq_s16(int16x8_t a, int16x8_t b);   // VHSUB.S16 q0,q0,q0
int32x4_t  vhsubq_s32(int32x4_t a, int32x4_t b);   // VHSUB.S32 q0,q0,q0
uint8x16_t vhsubq_u8(uint8x16_t a, uint8x16_t b);  // VHSUB.U8 q0,q0,q0
uint16x8_t vhsubq_u16(uint16x8_t a, uint16x8_t b); // VHSUB.U16 q0,q0,q0
uint32x4_t vhsubq_u32(uint32x4_t a, uint32x4_t b); // VHSUB.U32 q0,q0,q0
```

高位半部分向量减法

```
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
```

高位半部分向量舍入减法

```
int8x8_t   vrsubhn_s16(int16x8_t a, int16x8_t b);   // VRSUBHN.I16 d0,q0,q0
int16x4_t  vrsubhn_s32(int32x4_t a, int32x4_t b);   // VRSUBHN.I32 d0,q0,q0
int32x2_t  vrsubhn_s64(int64x2_t a, int64x2_t b);   // VRSUBHN.I64 d0,q0,q0
uint8x8_t  vrsubhn_u16(uint16x8_t a, uint16x8_t b); // VRSUBHN.I16 d0,q0,q0
uint16x4_t vrsubhn_u32(uint32x4_t a, uint32x4_t b); // VRSUBHN.I32 d0,q0,q0
uint32x2_t vrsubhn_u64(uint64x2_t a, uint64x2_t b); // VRSUBHN.I64 d0,q0,q0
```

E.3.4 比较

提供一系列比较内在函数。如果对于一条向量线比较结果为 true，则该向量线的结果为将所有位设置为一。如果对于一条向量线比较结果为 false，则将所有位设置为零。返回类型是无符号整数类型。这意味着可以将比较结果用作 vbsl 内在函数的第一个参数。

向量比较等于

```
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
```

向量比较大于或等于

```
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
```

向量比较小于或等于

```
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
```

向量比较大于

```
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
```

向量比较小于

```
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
```

向量绝对值比较大于或等于

```
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
```

向量绝对值比较小于或等于

```
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
```

向量绝对值比较大于

```
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
```

向量绝对值比较小于

```
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
```

向量测试位

```
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 差值绝对值

以下内在函数提供包含差值绝对值的运算。

参数间的差值绝对值：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
```

差值绝对值 - 长型

```
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
```

差值绝对值累加：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
```

差值绝对值累加 - 长型

```
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 最大值/最小值

以下内在函数提供最大值和最小值运算。

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 按对加

以下内在函数提供按对加运算。

按对加

```
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.I8 d0,d0,d0
uint16x4_t  vpadd_u16(uint16x4_t a, uint16x4_t b);   // VPADD.I16 d0,d0,d0
uint32x2_t  vpadd_u32(uint32x2_t a, uint32x2_t b);   // VPADD.I32 d0,d0,d0
float32x2_t vpadd_f32(float32x2_t a, float32x2_t b); // VPADD.F32 d0,d0,d0
```

长型按对加

```
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
```

长型按对加并累加

```
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 折叠最大值

vpmax -> 获取相邻对的最大值

```
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 折叠最小值

vpmin -> 获取相邻对的最小值

```
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   vpmin_u8(uint8x8_t a, uint8x8_t b);      // VPMIN.U8 d0,d0,d0
uint16x4_t  vpmin_u16(uint16x4_t a, uint16x4_t b);   // VPMIN.U16 d0,d0,d0
uint32x2_t  vpmin_u32(uint32x2_t a, uint32x2_t b);   // VPMIN.U32 d0,d0,d0
float32x2_t vpmin_f32(float32x2_t a, float32x2_t b); // VPMIN.F32 d0,d0,d0
```

E.3.10 倒数/平方根

近似倒数/步进和 1/近似平方根/步进

```
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 按有符号变量移位

以下内在函数提供包含按有符号变量移位的运算。

向量左移：Vr[i] := Va[i] << Vb[i] （负值右移）

```
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
```

向量饱和左移：（负值右移）

```
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
```

向量舍入左移：（负值右移）

```
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
```

向量饱和舍入左移：（负值右移）

```
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 按常数移位

以下内在函数提供按常数移位的运算。

向量按常数右移

```
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
```

向量按常数左移

```
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
```

向量舍入按常数右移

```
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
```

向量按常数右移并累加

```
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
```

向量舍入按常数右移并累加

```
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
```

向量饱和按常数左移

```
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
```

向量有符号->无符号饱和按常数左移

```
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
```

向量窄型饱和按常数右移

```
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.I16 d0,q0,#8
uint16x4_t vshrn_n_u32(uint32x4_t a, __constrange(1,16) int b); // VSHRN.I32 d0,q0,#16
uint32x2_t vshrn_n_u64(uint64x2_t a, __constrange(1,32) int b); // VSHRN.I64 d0,q0,#32
```

向量有符号->无符号窄型饱和按常数右移

```
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
```

向量有符号->无符号舍入窄型饱和按常数右移

```
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
```

向量窄型饱和按常数右移

```
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
```

向量舍入窄型按常数右移

```
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
```

向量舍入窄型饱和按常数右移

```
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
```

向量扩大按常数左移

```
int16x8_t  vshll_n_s8(int8x8_t a, __constrange(0,8) int b);     // VSHLL.S8 q0,d0,#0
int32x4_t  vshll_n_s16(int16x4_t a, __constrange(0,16) int b);  // VSHLL.S16 q0,d0,#0
int64x2_t  vshll_n_s32(int32x2_t a, __constrange(0,32) int b);  // VSHLL.S32 q0,d0,#0
uint16x8_t vshll_n_u8(uint8x8_t a, __constrange(0,8) int b);    // VSHLL.U8 q0,d0,#0
uint32x4_t vshll_n_u16(uint16x4_t a, __constrange(0,16) int b); // VSHLL.U16 q0,d0,#0
uint64x2_t vshll_n_u32(uint32x2_t a, __constrange(0,32) int b); // VSHLL.U32 q0,d0,#0
```

E.3.13 移位并插入

以下内在函数提供包含移位并插入的运算。

向量右移并插入

```
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
```

向量左移并插入

```
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 加载并存储单个向量

加载并存储某类型的单个向量。

```
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]
float16x8_t vld1q_f16(__transfersize(8) __fp16 const * ptr);
                                                  // VLD1.16 {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]
float16x4_t vld1_f16(__transfersize(4) __fp16 const * ptr);
                                                // VLD1.16 {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]
float16x4_t vld1q_lane_f16(__transfersize(1) __fp16 const * ptr, float16x4_t vec, __constrange(0,3) int
lane);
```

```
                                                    // VLD1.16 {d0[0]}, [r0]
float16x8_t vld1q_lane_f16(__transfersize(1) __fp16 const * ptr, float16x8_t vec, __constrange(0,7) int
lane);
                                                    // VLD1.16 {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]
float16x8_t vld1q_dup_f16(__transfersize(1) __fp16 const * ptr);
                                                // VLD1.16 {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]
float16x4_t vld1_dup_f16(__transfersize(1) __fp16 const * ptr);
                                                // VLD1.16 {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_f16(__transfersize(8) __fp16 * ptr, float16x8_t val);
                                                // VST1.16 {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_f16(__transfersize(4) __fp16 * ptr, float16x4_t val);
                                                // VST1.16 {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_f16(__transfersize(1) __fp16 * ptr, float16x8_t val, __constrange(0,7) int lane);
                                          // VST1.16 {d0[0]}, [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_f16(__transfersize(1) __fp16 * ptr, float16x4_t val, __constrange(0,3) int lane);
                                          // VST1.16 {d0[0]}, [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 加载并存储 N 元素结构

以下内在函数加载或存储 *n*-元素结构。数组结构的定义方式类似，例如 int16x4x2_t 结构定义如下：

```
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]
float16x8x2_t vld2q_f16(__transfersize(16) __fp16 const * ptr);
                                                  // VLD2.16 {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]
float16x4x2_t vld2_f16(__transfersize(8) __fp16 const * ptr);
                                                  // VLD2.16 {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]
float16x8x3_t vld3q_f16(__transfersize(24) __fp16 const * ptr);
                                                // VLD3.16 {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]
float16x4x3_t vld3_f16(__transfersize(12) __fp16 const * ptr);
                                                // VLD3.16 {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]
float16x8x4_t vld4q_f16(__transfersize(32) __fp16 const * ptr);
                                                // VLD4.16 {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]
float16x4x4_t vld4_f16(__transfersize(16) __fp16 const * ptr);
                                                // VLD4.16 {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]
float16x4x2_t vld2_dup_f16(__transfersize(2) __fp16 const * ptr);
                                                // VLD2.16 {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]
float16x4x3_t vld3_dup_f16(__transfersize(3) __fp16 const * ptr);
                                                // VLD3.16 {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]
float16x4x4_t vld4_dup_f16(__transfersize(4) __fp16 const * ptr);
                                                // VLD4.16 {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]
float16x8x2_t vld2q_lane_f16(__transfersize(2) __fp16 const * ptr, float16x8x2_t src, __constrange(0,7)
int lane);
                                                // VLD2.16 {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]
float16x4x2_t vld2_lane_f32(__transfersize(2) __fp16 const * ptr, float16x4x2_t src, __constrange(0,3)
int lane);
                                                // VLD2.16 {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]
float16x8x3_t vld3q_lane_f32(__transfersize(3) __fp16 const * ptr, float16x8x3_t src, __constrange(0,7)
int lane);
                                                // VLD3.16 {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]
float16x4x3_t vld3_lane_f16(__transfersize(3) __fp16 const * ptr, float16x4x3_t src, __constrange(0,3)
int lane);
                                                    // VLD3.16 {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]
float16x8x4_t vld4q_lane_f32(__transfersize(4) __fp16 const * ptr, float16x8x4_t src, __constrange(0,7)
int lane);
                                                    // VLD4.16 {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]
float16x4x4_t vld4_lane_f16(__transfersize(4) __fp16 const * ptr, float16x4x4_t src, __constrange(0,3)
int lane);
                                                // VLD4.16 {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_f16(__transfersize(16) __fp16 * ptr, float16x8x2_t val);
                                                // VST2.16 {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_f16(__transfersize(8) __fp16 * ptr, float16x4x2_t val);
                                                // VST2.16 {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_f16(__transfersize(24) __fp16 * ptr, float16x8x3_t val);
                                                // VST3.16 {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_f16(__transfersize(12) __fp16 * ptr, float16x4x3_t val);
                                          // VST3.16 {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_f16(__transfersize(32) __fp16 * ptr, float16x8x4_t val);
                                          // VST4.16 {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_f16(__transfersize(16) __fp16 * ptr, float16x4x4_t val);
```

```
                                          // VST4.16 {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_f16(__transfersize(2) __fp16 * ptr, float16x8x2_t val, __constrange(0,7) int lane);
                                          // VST2.16 {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_f16(__transfersize(2) __fp16 * ptr, float16x4x2_t val, __constrange(0,3) int lane);
                                          // VST2.16 {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_f16(__transfersize(3) __fp16 * ptr, float16x8x3_t val, __constrange(0,7) int lane);
                                                // VST3.16 {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_f16(__transfersize(3) __fp16 * ptr, float16x4x3_t val, __constrange(0,3) int lane);
                                                // VST3.16 {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_f16(__transfersize(4) __fp16 * ptr, float16x8x4_t val, __constrange(0,7) int lane);
                                                // VST4.16 {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_f16(__transfersize(4) __fp16 * ptr, float16x4x4_t val, __constrange(0,3) int lane);
                                                // VST4.16 {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 从向量提取向量线

以下内在函数从向量提取单条向量线（元素）。

```
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 在向量内设置向量线

以下内在函数在向量内设置单条向量线（元素）。

```
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 从位模式初始化向量

以下内在函数从文字位模式创建向量。

```
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
float16x4_t vcreate_f16(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 将所有向量线设置为相同的值

以下内在函数将所有向量线设置为相同的值。

将所有向量线设置为相同的值

```
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
```

将向量的所有向量线设置为一条向量线的值

```
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 组合向量

以下内在函数将两个 64 位向量组合为单个 128 位向量。

```
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
float16x8_t vcombine_f16(float16x4_t low, float16x4_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 拆分向量

以下内在函数将一个 128 位向量拆分为 2 个 64 位向量。

```
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
float16x4_t vget_high_f16(float16x8_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
float16x4_t vget_low_f16(float16x8_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 转换向量

以下内在函数用于转换向量。

从浮点转换

```
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
```

转换为浮点

```
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
```

在浮点之间转换

```
float16x4_t vcvt_f16_f32(float32x4_t a); // VCVT.F16.F32 d0, q0
float32x4_t vcvt_f32_f16(float16x4_t a); // VCVT.F32.F16 q0, d0
```

向量窄型整数

```
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
```

向量长移

```
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
```

向量饱和窄型整数

```
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
```

向量饱和窄型整数有符号->无符号的转换

```
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 表查找

```
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 扩展表查找内在函数

```
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 针对标量值的运算

仅当标量参数为常数或用于 vget_lane 内在函数之一时，才能保证以下内在函数生成有效的代码。

向量与标量进行的乘加

```
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]
```

向量与标量进行的扩大乘加

```
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]
```

向量与标量进行的扩大饱和加倍乘加

```
int32x4_t   vqdmlal_lane_s16(int32x4_t a, int16x4_t b, int16x4_t v, __constrange(0,3) int l);
                                                  // VQDMLAL.S16 q0, d0, d0[0]
int64x2_t   vqdmlal_lane_s32(int64x2_t a, int32x2_t b, int32x2_t v, __constrange(0,1) int l);
                                                  // VQDMLAL.S32 q0, d0, d0[0]
```

向量与标量进行的乘减

```
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]
```

向量与标量进行的扩大乘减

```
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]
```

向量与标量进行的扩大饱和加倍乘减

```
int32x4_t   vqdmlsl_lane_s16(int32x4_t a, int16x4_t b, int16x4_t v, __constrange(0,3) int l);
                                          // VQDMLSL.S16 q0, d0, d0[0]
int64x2_t   vqdmlsl_lane_s32(int64x2_t a, int32x2_t b, int32x2_t v, __constrange(0,1) int l);
                                          // VQDMLSL.S32 q0, d0, d0[0]
```

向量乘以标量

```
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]
```

向量与标量进行的长型乘法

```
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]
```

向量与标量进行的长型乘法

```
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]
```

向量与标量进行的饱和加倍长型乘法

```
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]
```

向量与标量进行的饱和加倍长型乘法

```
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]
```

向量与标量进行的高位饱和加倍乘法

```
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]
```

向量与标量进行的高位饱和加倍乘法

```
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]
```

向量与标量进行的高位饱和舍入加倍乘法

```
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]
```

向量与标量进行的高位舍入饱和加倍乘法

```
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]
```

向量与标量进行的乘加

```
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]
```

向量与标量进行的扩大乘加

```
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]
```

向量与标量进行的扩大饱和加倍乘加

```
int32x4_t   vqdmlal_n_s16(int32x4_t a, int16x4_t b, int16_t c);    // VQDMLAL.S16 q0, d0, d0[0]
int64x2_t   vqdmlal_n_s32(int64x2_t a, int32x2_t b, int32_t c);    // VQDMLAL.S32 q0, d0, d0[0]
```

向量与标量进行的乘减

```
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]
```

向量与标量进行的扩大乘减

```
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]
```

向量与标量进行的扩大饱和加倍乘减

```
int32x4_t   vqdmlsl_n_s16(int32x4_t a, int16x4_t b, int16_t c);   // VQDMLSL.S16 q0, d0, d0[0]
int64x2_t   vqdmlsl_n_s32(int64x2_t a, int32x2_t b, int32_t c);   // VQDMLSL.S32 q0, d0, d0[0]
```

E.3.26 向量提取

```
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 反转向量元素 （交换端标记）

VREVn.m 反转 n 位宽的集合内 m 位向量线的顺序。

```
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 其他单操作数算法

以下内在函数提供其他单操作数算法。

绝对值：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
```

饱和绝对值：Vd[i] = 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
```

求反：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
```

饱和求反：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
```

计算前导符号位数目

```
int8x8_t  vcls_s8(int8x8_t a);     // VCLS.S8 d0,d0
int16x4_t vcls_s16(int16x4_t a);   // VCLS.S16 d0,d0
int32x2_t vcls_s32(int32x2_t a);   // VCLS.S32 d0,d0
int8x16_t vclsq_s8(int8x16_t a);   // VCLS.S8 q0,q0
int16x8_t vclsq_s16(int16x8_t a);  // VCLS.S16 q0,q0
int32x4_t vclsq_s32(int32x4_t a);  // VCLS.S32 q0,q0
```

计算前导零数目

```
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
```

计算设置位数

```
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
```

近似倒数

```
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
```

近似平方根倒数

```
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 逻辑运算

以下内在函数提供按位逻辑运算。

按位非

```
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
```

按位与

```
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
```

按位或

```
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
```

按位异或 （EOR 或 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
```

位清零

```
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
```

按位或补

```
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  vornq_s16(int16x8_t a, int16x8_t b);    // VORN q0,q0,q0
int32x4_t  vornq_s32(int32x4_t a, int32x4_t b);    // VORN q0,q0,q0
int64x2_t  vornq_s64(int64x2_t a, int64x2_t b);    // VORN q0,q0,q0
uint8x16_t vornq_u8(uint8x16_t a, uint8x16_t b);   // VORN q0,q0,q0
uint16x8_t vornq_u16(uint16x8_t a, uint16x8_t b);  // VORN q0,q0,q0
uint32x4_t vornq_u32(uint32x4_t a, uint32x4_t b);  // VORN q0,q0,q0
uint64x2_t vornq_u64(uint64x2_t a, uint64x2_t b);  // VORN q0,q0,q0
```

按位选择

注意

此内在函数可以根据寄存器的分配编译为 VBSL/VBIF/VBIT 中的任意一个。

```
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 转置运算

这些内在函数提供转置运算。

转置元素

```
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
```

交叉存取元素

```
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
```

反向交叉存取元素

```
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 向量重新解释类型转换运算

在某些情况下，可能希望将向量视为另一类型而不更改其值。提供了一组内在函数以执行此类型的转换。

语法

`vreinterpret{q}_dsttype_srctype`

其中：

`q` 指定对 128 位向量执行转换。如果不存在 128 位向量，则对 64 位向量执行转换。

`dsttype` 表示要转换成的类型。

`srctype` 表示被转换的类型。

示例

以下内在函数将四个有符号 16 位整数组成的向量重新解释为四个无符号整数组成的向量：

```
uint16x4_t vreinterpret_u16_s16(int16x4_t a);
```

以下内在函数将四个 32 位浮点值整数组成的向量重新解释为四个有符号整数组成的向量。

```
int8x16_t vreinterpretq_s8_f32(float32x4_t a);
```

这些转换不会更改向量所表示的位模式。