RealView® 编译工具

4.0 版

《编译器用户指南》

ARM®

ARM DUI 0205IC (ID7/8/09)

RealView 编译工具
《编译器用户指南》

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	更改
2002 年 8 月	A	非保密	1.2 版
2003 年 1 月	B	非保密	2.0 版
2003 年 9 月	C	非保密	RealView Developer Suite v2.0 2.0.1 版
2004 年 1 月	D	非保密	RealView Developer Suite v2.1 2.1 版
2004 年 12 月	E	非保密	RealView Developer Suite v2.2 2.2 版
2005 年 5 月	F	非保密	RealView Developer Suite v2.2 SP1 2.2 版
2006 年 3 月	G	非保密	RealView Development Suite v3.0 3.0 版
2007 年 3 月	H	非保密	RealView Development Suite v3.1 3.1 版
2008 年 9 月	I	非保密	RealView Development Suite v4.0 4.0 版
2009 年 1 月 23 日	I	非保密	RealView Development Suite 4.0 版的文档更新

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
编译器用户指南

前言

本前言介绍 《RealView 编译工具编译器用户指南》。本章分为以下几节：

关于本手册

本手册为用户提供有关 *RealView 编译工具* (RVCT) 的信息，并概述了 ARM 编译器和 NEON™ 向量化编译器支持的命令行选项和编译器特有的功能。

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 ***ARM 编译器使用入门***

阅读本章后，可以大致了解命令行选项和编译器特定的功能。本章介绍如何调用编译器、如何将选项传递给其他 RVCT 工具，以及如何控制诊断消息。

第 3 章 ***使用 NEON 向量化编译器***

本章提供有关 NEON 向量化编译器的教程。本章介绍 NEON 单元，并说明如何利用自动向量化功能。

第 4 章 ***编译器功能***

阅读本章后，可以大致了解 ARM 编译器支持的内在函数。

第 5 章 ***编程惯例***

阅读本章后，可以大致了解 RVCT 中的良好的编程惯例。

第 6 章 ***诊断消息***

阅读本章后，可以大致了解 RVCT 工具生成的诊断消息。

第 7 章 ***使用内联汇编器和嵌入式汇编器***

阅读本章后，可以了解 ARM 编译器提供的内联汇编器和嵌入式汇编器。

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 编译器参考指南》* (ARM DUI 0348)
- *《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)

*《RVDS 入门指南》*中提供了一个术语表。

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)
- 《ARMv7 体系结构参考手册》 (ARM DDI 0403)
- 《ARMv6-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

本手册的目的不是介绍 C 或 C++ 编程语言，并不致力于讲解用 C 或 C++ 编程，也不是一本 C 或 C++ 标准的参考手册。有关编程的信息，请参考其他书籍。

以下出版物介绍 C++ 语言：

- *《ISO/IEC 14882:2003, C++ 标准》*。
- *《C++ 编程语言》*， Stroustrup, B. 著 （1997 年第 3 版）。Addison-Wesley Publishing Company, Reading, Massachusetts。ISBN 0-201-88954-4。

以下书籍提供一般 C++ 编程信息：

- *《C++ 的设计和演进》*， Stroustrup, B. 著 （1994 年）。Addison-Wesley Publishing Company, Reading, Massachusetts。ISBN 0-201-54330-3。

  此书介绍 C++ 如何从其最初的设计发展到今天使用的语言。
- Vandevoorde, D 和 Josuttis, N.M. 合著 《C++ 模板：完全指南》 (*C++ Templates: The Complete Guide*) （2003 年）。Addison-Wesley Publishing Company, Reading, Massachusetts。ISBN 0-201-73484-2。
- *《高效 C++》*， Meyers, S. 著 （1992 年）。Addison-Wesley Publishing Company, Reading, Massachusetts。ISBN 0-201-56364-9。

  此书提供关于 C++ 有效开发的简短而具体的准则。

- *《更有效的 C++》*， Meyers, S. 著 （1997 年第 2 版）。Addison-Wesley Publishing Company, Reading, Massachusetts。ISBN 0-201-92488-9。

以下出版物提供一般 C 编程信息：

- ISO/IEC 9899:1999, *《C 标准》*。

  此标准可从国家/地区级的标准机构 （例如，法国的 AFNOR 和美国的 ANSI）获得。

- *《C 编程语言》*， Kernighan, B.W. 和 Ritchie, D.M. 著 （1988 年第 2 版）。Prentice-Hall, Englewood Cliffs, NJ, USA。ISBN 0-13-110362-8。

  本书由 C 语言的原始设计者与实现者合著，并针对 ANSI C 的重要内容进行了更新。

- *《C 参考手册》*， Harbison, S.P. 和 Steele, G.L. 著 （2002 年第 5 版）。Prentice-Hall, Englewood Cliffs, NJ, USA。ISBN 0-13-089592-X。

  此书为一本关于 C 的非常全面的参考手册，包括关于 ANSI C 的有用信息。

- *《标准 C 库》*， Plauger, P. 著 （1991 年）。Prentice-Hall, Englewood Cliffs, NJ, USA。ISBN 0-13-131509-9。

  此书全面介绍了 C 库的 ANSI 和 ISO 标准的处理。

- *《C 陷进》*， Koenig, A. 著。 Addison-Wesley (1989), Reading, Mass。ISBN 0-201-17928-8。

  此书介绍如何在 C 编程中避免最常见的陷阱。它提供了适于所有级别的 C 编程水平的丰富阅读材料。

有关*使用任意记录格式进行的调试* (DWARF) 调试表标准以及*可执行和链接格式* (ELF) 规范的最新信息，请访问 `http://www.dwarfstd.org`。

以下出版物提供有关*欧洲电信标准协会* (ETSI) 基本操作的信息。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)*)。

德州仪器公司网站 `http://www.ti.com` 提供有关德州仪器编译器内在函数信息的出版物。

反馈

ARM Limited 欢迎您提出关于 RVCT 及文档的反馈信息。

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 RVCT 附带的 ARM® 编译器。还介绍遵循的标准，并概述 RVCT 附带的运行时库。本章分为以下几节：

1.1 关于 ARM 编译器

ARM 编译器 armcc 是一种 C 和 C++ 优化编译器，可将标准 C 和标准 C++ 源代码编译为适用于基于 ARM 体系结构的处理器的机器代码。该编译器遵循《ARM 体系结构的基础标准应用程序二进制接口》 (*Base Standard Application Binary Interface for the ARM Architecture*) (BSABI)，生成支持 DWARF 3 调试表的 ELF 格式的输出对象文件。它使用 *Edison Design Group* (EDG) 前端。

如果要从以前版本升级到 RVCT，或者是初次使用 RVCT，请务必阅读 *《RVCT 要点指南》*以了解最新信息。

1.2 关于 NEON 向量化编译器

NEON™ 是 ARM 高级单指令多数据 (SIMD) 扩展的实现。

RVCT 提供 `armcc --vectorize` （ARM 编译器的向量化模式），它面向具有 NEON 单元的 ARM 处理器 （如 Cortex-A8 和 Cortex-A9）。

注意

若要针对 Cortex-A9 目标进行编译，必须拥有 RealView Development Suite Professional 的许可证。

向量化意味着编译器直接从 C 或 C++ 代码生成 NEON 向量指令。

作为自动编译器向量化的替换方式， RVCT 还支持 NEON 内在函数作为向量化编译器与编写汇编代码之间的 SIMD 代码生成的中间步骤。

请参阅：

- 第 3 章 *使用 NEON 向量化编译器*
- 第4-11 页的*NEON 内在函数*
- 《编译器参考指南》中附录 E *使用 NEON 支持*

1.3 源语言模式

ARM 编译器提供了三种不同的源语言模式，可用于编译不同版本的 C 和 C++ 源代码：

ISO C90 ARM 编译器根据 1990 C 标准和附录的定义编译 C 代码。

使用编译器选项 `--c90` 可编译 C90 代码。这是缺省设置。

ISO C99 ARM 编译器根据 1999 C 标准和附录的定义编译 C 代码。

使用编译器选项 `--c99` 可编译 C99 代码。

ISO C++ ARM 编译器根据 2003 标准中的定义编译 C++ 代码，不过宽流和导出模板除外。

要编译 C++ 代码，可使用编译器选项 `--cpp`。

编译器支持大量 C 及 C++ 语言扩展。例如，支持某些 GNU 编译器扩展。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。

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。

要在 GNU 模式下进行编译，请使用编译器选项 `--gnu`。

有关源语言模式和语言遵从性的详细信息，请参阅第5-43 页的*C99 的新功能*。此外，请参阅：

- 《编译器参考指南》中第1-3 页的*源语言模式*
- 《编译器参考指南》中第1-5 页的*语言扩展和语言遵从性*
- *《编译器参考指南》*中第2-22 页的*--c90*
- *《编译器参考指南》*中第2-22 页的*--c99*
- *《编译器参考指南》*中第2-30 页的*--cpp*
- *《编译器参考指南》*中第2-66 页的*--gnu*
- *《编译器参考指南》*中第2-116 页的*--strict*, *--no_strict*。

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。

ARM 库符合：

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

请参阅：

- 《编译器参考指南》中第1-7 页的*C 和 C++ 库*
- *《库指南》*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`。它还包括称为*标准模板库* (STL) 的数据结构和算法。

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。对于主要生成选项的每一种组合 （如目标系统的字节顺序、是否选择交互操作以及是否选择浮点支持），都有对应的 C 和 C++ 库。

请参阅 《库指南》中第 2 章 *C 和 C++ 库*。

第 2 章
ARM 编译器使用入门

本章概括了 ARM 编译器 armcc 接受的命令行选项，本章介绍如何调用编译器、如何将选项传递给其他 RVCT 工具，以及如何控制诊断消息。本章分为以下几节：

请参阅 *《编译器参考指南》*。

2.1 使用命令行选项

使用命令行选项可以控制编译器操作的许多方面。

根据选项类型而定，以下规则可能适用：

单字母选项

在所有单字母选项或带参数的单字母选项之前，都有一个单短划线 -。选项与参数之间可以有空格，参数也可以紧跟在选项之后。例如：

`-J directory`

`-Jdirectory`

关键字选项

在所有关键字选项或带参数的关键字选项前面，都有一个双短划线 --。选项和参数之间需要用 = 或空格字符分隔。例如：

```
--depend=file.d
```

```
--depend file.d
```

包含非前置 - 或 _ 的编译器选项可以使用这两种字符之一。例如，`--force_new_nothrow` 与 `--force-new-nothrow` 相同。

要编译名称以短划线开头的文件，请使用 POSIX 选项 -- 指定所有的后续参数均视为文件名，而不是命令开关。例如，要编译名为 `-ifile_1` 的文件，可以使用以下命令：

```
armcc -c -- -ifile_1
```

2.1.1 调用 ARM 编译器

调用 ARM 编译器的命令是：

armcc [*help-options*] [*source-language*] [*search-paths*] [*project-template-options*] [*PCH-options*] [*preprocessor-options*] [*C++-language*] [*output-format*] [*target-options*] [*debug-options*] [*code-generation-options*] [*optimization-options*] [*diagnostic-options*] [*additional-checks*] [*PCS-options*] [*pass-thru-options*] [*arm-linux-options*] [*source*]

有关以下选项的详细信息，请参阅 《编译器参考指南》中的第 2 章 *编译器命令行选项*：

help-options 显示主要命令行选项、编译器的版本号，以及编译器处理命令行的方式：

- 第2-69 页的*--help*
- 第2-112 页的*--show_cmdline*
- 第2-130 页的*--vsn*

source-language 指定编译器接受的源语言版本：

- 第2-22 页的*--c90*
- 第2-22 页的*--c99*
- 第2-23 页的*--compile_all_input*, *--no_compile_all_input*
- 第2-30 页的*--cpp*
- 第2-66 页的*--gnu*
- 第2-116 页的*--strict*, *--no_strict*
- 第2-117 页的*--strict_warnings*

可以结合使用这些语言选项。例如：

```
armcc --c90 --gnu
```

search-paths 指定搜索包含文件的目录：

- 第2-69 页的*-Idir[,dir,...]*
- 第2-76 页的*-Jdir[,dir,...]*
- 第2-76 页的*--kandr_include*
- 第2-103 页的*--preinclude=filename*
- 第2-106 页的*--reduce_paths*, *--no_reduce_paths*
- 第2-118 页的*--sys_include*

有关如何结合使用这些选项的详细信息，请参阅第2-14 页的*头文件*。

project-template-options

控制项目模板的行为：

- 第2-105 页的*--project=filename*, *--no_project=filename*
- 第2-107 页的*--reinitialize_workdir*
- 第2-133 页的*--workdir=directory*

PCH-options

控制 PCH 文件的处理：

- 第2-34 页的*--create_pch=filename*
- 第2-98 页的*--pch*
- 第2-99 页的*--pch_dir=dir*
- 第2-100 页的*--pch_messages*, *--no_pch_messages*
- 第2-100 页的*--pch_verbose*, *--no_pch_verbose*
- 第2-126 页的*--use_pch=filename*

preprocessor-options

指定预处理器行为，包括预处理器输出和宏定义：

- 第2-21 页的*-C*
- 第2-22 页的*--code_gen*, *--no_code_gen*
- 第2-35 页的*-Dname[(parm-list)][=def]*
- 第2-51 页的*-E*
- 第2-86 页的*-M*
- 第2-124 页的*-Uname*

C++-language

指定 C++ 编译特有的选项：

- 第2-3 页的*--anachronisms*, *--no_anachronisms*
- 第2-38 页的*--dep_name*, *--no_dep_name*
- 第2-54 页的*--export_all_vtbl*, *--no_export_all_vtbl*
- 第2-57 页的*--force_new_nothrow*, *--no_force_new_nothrow*
- 第2-64 页的*--friend_injection*, *--no_friend_injection*
- 第2-68 页的*--guiding_decls*, *--no_guiding_decls*

- 第2-71 页的*--implicit_include*, *--no_implicit_include*
- 第2-71 页的*--implicit_include_searches*, *--no_implicit_include_searches*
- 第2-72 页的*--implicit_typename*, *--no_implicit_typename*
- 第2-92 页的*--nonstd_qualifier_deduction*, *--no_nonstd_qualifier_deduction*
- 第2-96 页的*--old_specializations*, *--no_old_specializations*
- 第2-98 页的*--parse_templates*, *--no_parse_templates*
- 第2-101 页的*--pending_instantiations=n*
- 第2-110 页的*--rtti*, *--no_rtti*
- 第2-127 页的*--using_std*, *--no_using_std*
- 第2-128 页的*--vfe*, *--no_vfe*

output-format

指定编译器的输出格式。使用这些选项，可以生成对象文件、汇编语言输出列表文件以及 make 文件相关性文件：

- 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2-41 页的*--depend_system_headers*, *--no_depend_system_headers*
- 第2-73 页的*--info=totals*
- 第2-75 页的*--interleave*
- 第2-80 页的*--list*
- 第2-87 页的*--md*
- 第2-92 页的*-o filename*
- 第2-111 页的*-S*
- 第2-115 页的*--split_sections*

target-options 指定目标处理器或体系结构以及启动时使用的目标指令集：

- 第2-8 页的*--arm*
- 第2-23 页的*--compatible=name*
- 第2-30 页的*--cpu=list*
- 第2-30 页的*--cpu=name*
- 第2-61 页的*--fpu=list*
- 第2-61 页的*--fpu=name*
- 第2-119 页的*--thumb*

请参阅第2-22 页的*指定目标处理器或体系结构*。

half-precision floating-point option

将半精度浮点数用作 VFPv3 体系结构的可选扩展：

- 第2-59 页的*--fp16_format=format*

debug-options 控制调试表的格式和生成：

- 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2-65 页的*-g*

code-generation-options

指定 ARM 编译器的代码生成选项，包括端标记、符号可见性和对齐标准：

- 第2-3 页的*--alternative_tokens*, *--no_alternative_tokens*
- 第2-17 页的*--bigend*
- 第2-20 页的*--bss_threshold=num*
- 第2-49 页的*--dllexport_all*, *--no_dllexport_all*
- 第2-50 页的*--dllimport_runtime*, *--no_dllimport_runtime*
- 第2-50 页的*--dollar*, *--no_dollar*
- 第2-52 页的*--enum_is_int*
- 第2-53 页的*--exceptions*, *--no_exceptions*

- 第2-54 页的*--exceptions_unwind*, *--no_exceptions_unwind*
- 第2-54 页的*--export_all_vtbl*, *--no_export_all_vtbl*
- 第2-55 页的*--export_defs_implicitly*, *--no_export_defs_implicitly*
- 第2-55 页的*--extended_initializers*, *--no_extended_initializers*
- 第2-69 页的*--hide_all*, *--no_hide_all*
- 第2-83 页的*--littleend*
- 第2-84 页的*--locale=lang_country*
- 第2-85 页的*--loose_implicit_cast*
- 第2-88 页的*--message_locale=lang_country[.codepage]*
- 第2-89 页的*--min_array_alignment=opt*
- 第2-90 页的*--multibyte_chars*, *--no_multibyte_chars*
- 第2-102 页的*--pointer_alignment=num*
- 第2-109 页的*--restrict*, *--no_restrict*
- 第2-113 页的*--signed_bitfields*, *--unsigned_bitfields*
- 第2-113 页的*--signed_chars*, *--unsigned_chars*
- 第2-114 页的*--split_ldm*
- 第2-125 页的*--unaligned_access*, *--no_unaligned_access*
- 第2-127 页的*--vectorize*, *--no_vectorize*
- 第2-130 页的*--vla*, *--no_vla*
- 第2-132 页的*--wchar16*
- 第2-132 页的*--wchar32*

optimization-options

控制代码优化的级别和类型：

- 第2-16 页的*--autoinline*, *--no_autoinline*
- 第2-36 页的*--data_reorder*, *--no_data_reorder*
- 第2-58 页的*--forceinline*
- 第2-59 页的*--fpmode=model*
- 第2-73 页的*--inline*, *--no_inline*
- 第2-77 页的*--library_interface=lib*
- 第2-79 页的*--library_type=lib*
- 第2-85 页的*--lower_ropi*, *--no_lower_ropi*
- 第2-85 页的*--lower_rwpi*, *--no_lower_rwpi*
- 第2-90 页的*--multifile*, *--no_multifile*
- 第2-94 页的*-Onum*
- 第2-96 页的*-Ospace*
- 第2-97 页的*-Otime*
- 第2-109 页的*--retain=option*

——注意——

优化标准可限制编译器生成的调试信息。

diagnostic-options

控制编译器输出的诊断消息：

- 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2-47 页的*--diag_suppress=optimizations*
- 第2-48 页的*--diag_warning=tag[,tag,...]*
- 第2-48 页的*--diag_warning=optimizations*
- 第2-52 页的*--errors=filename*
- 第2-108 页的*--remarks*
- 第2-130 页的*-W*

- 第2-134 页的 *--wrap_diagnostics*, *--no_wrap_diagnostics*

请参阅第 6 章 *诊断消息*。

command-line option file

指定包含其他命令行选项的文件：

- 第2-129 页的 *--via=filename*

multiple compilations

指定包含有关上次编译的信息的反馈文件：

- 第2-56 页的 *--feedback=filename*
- 第2-104 页的 *--profile=filename*

PCS-options 指定要使用的过程调用标准：

- 第2-4 页的 *--apcs=qualifer...qualifier*

请参阅第2-23 页的 *指定过程调用标准 (AAPCS)*。

pass-thru-options 指示编译器将选项传递给其他 RVCT 工具：

- 第2-2 页的 *-Aopt*
- 第2-77 页的 *-Lopt*

arm-linux-options 指定选项以配置 RVCT 用于 ARM Linux，以及生成面向 ARM Linux 的应用程序和共享库：

- 第2-11 页的 *--arm_linux_configure*
- 第2-10 页的 *--arm_linux_config_file=path*
- 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2-8 页的 *--arm_linux*
- 第2-13 页的 *--arm_linux_paths*
- 第2-111 页的 *--shared*
- 第2-121 页的 *--translate_gcc*

- 第2-119 页的*--translate_g++*
- 第2-122 页的*--translate_gld*

source 提供包含 C 或 C++ 源代码的一个或多个文本文件的文件名。缺省情况下，编译器在当前目录中查找源文件并创建输出文件。

如果源文件是汇编文件，即扩展名为 .s，则编译器会激活 ARM 汇编器来处理源文件。

此选项不用于 *arm-linux-options*。对于所有其他选项，则是强制性的。

ARM 编译器接受一个或多个输入文件，例如：

```
armcc -c [options] ifile_1 ... ifile_n
```

如果为输入文件指定短划线 -，则会使编译器从 stdin 中读取。要指定将所有后续参数均视为文件名，而不是命令开关，请使用 POSIX 选项 --。请参阅第2-2 页的*使用命令行选项*。

缺省行为

编译器启动配置由编译器根据指定的命令行选项和文件扩展名来确定。命令行选项可以覆盖由文件扩展名确定的缺省配置。编译器启动语言可以是 C 或 C++，而指令集可以是 ARM 或 Thumb。

使用单个命令编译多个文件时，所有文件的类型必须相同，即 C 或 C++。编译器不能根据文件扩展名切换语言。在下例中，由于指定源文件的语言不同而产生了错误：

```
armcc -c test1.c test2.cpp
```

如果指定的文件的扩展名发生冲突，可以强制编译器针对 C 或 C++ 编译这两个文件，而不考虑文件扩展名。例如：

```
armcc -c --cpp test1.c test2.cpp
```

如果有以 .c 开头的无法识别的扩展名 （例如 *filename*.cmd），则会生成错误消息。

在一次编译中指定多个源文件时，不支持处理*预编译头* (PCH) 文件。如果请求 PCH 处理并且指定多个主源文件，则编译器会发出一条错误消息并中止编译。

请参阅第2-17 页的*预编译的头文件*。

2.1.2 命令行选项排序

通常，在一次编译器调用中，命令行选项可以按任意顺序显示。但是，一些选项的效果取决于它们在命令行中出现的顺序，以及与其他相关选项（例如，前缀为 -O 的优化选项或 PCH 选项）结合使用的方式。请参阅第2-17 页的*预编译的头文件*。

在编译器中可以使用多个选项，即便这些选项可能冲突，也是如此。这就是说，可以将新的选项附加至现有的命令行，例如 make 或 via 文件。

如果同一行中的选项覆盖前面的选项，则最后找到的选项总是优先执行。例如：

```
armcc -O1 -O2 -Ospace -Otime ...
```

由编译器以如下方式执行：

```
armcc -O2 -Otime
```

要查看编译器是如何处理命令行的，请使用 `--show_cmdline` 选项。此选项显示编译器使用的非缺省选项。任何 via 文件的内容都会展开。在此处使用的示例中，尽管编译器执行 `armcc -O2 -Otime`，`--show_cmdline` 的输出不包括 `-O2`。这是因为 `-O2` 是缺省优化级别，而 `--show_cmdline` 不显示在缺省情况下应用的选项。

2.1.3 使用环境变量指定命令行选项

通过设置 `RVCT40_CCOPT` 环境变量的值，可以指定命令行选项。其语法与命令行语法相同。编译器读取 `RVCT40_CCOPT` 的值，然后将其插入到命令字符串前面。也就是说，在 `RVCT40_CCOPT` 中指定的选项可由命令行中的参数覆盖。

2.1.4 自动完成命令行选项

您可以选择请求自动完成命令行选项。为此，可在要自动完成的字符后面放置点 (.)。自动完成仅适用于关键字选项。

参数必须用等号 (=) 字符或空格字符与点分开。不能对选项的参数使用自动完成。

必须包含足够的字符以使自动完成选项是唯一的。例如，使用 `--diag_su.=223` 可在命令行中指定 `--diag_suppress=223`。

请参阅第2-2 页的*使用命令行选项*。

2.1.5 从文件读取编译器选项

如果操作系统限制命令行的长度，则可以使用以下编译器选项在文件中提供附加的命令行选项：

`--via` *filename*

编译器将打开指定文件，并从中读取附加命令行选项。

请参阅 *《编译器参考指南》*中附录 A *via 文件语法*。

2.1.6 指定 stdin 输入

使用减号 (-) 作为源文件名可指示编译器接受来自 `stdin` 的输入。缺省的编译器模式为 C。

要终止输入，请执行以下操作：

- 在 Microsoft Windows 系统上先按 `Ctrl-Z` 然后按 `Return` 键。
- 在 Red Hat Linux 系统上按 `Ctrl-D` 键。

如果满足下面两个条件，则在终止输入后，会向输出流发送键盘输入的汇编列表：

- 没有指定输出文件
- 没有指定仅限预处理器的选项，例如 `-E`。

如果使用 `-o` 选项指定输出文件，则写入对象文件。如果指定 `-E` 选项，则会将预处理器输出发送到输出流。如果指定 `-o-` 选项，则将输出发送到 `stdout`。

2.2 文件命名约定

ARM 编译器使用文件名后缀来标识编译和链接阶段涉及的文件类。表 2-1 中介绍了编译器可识别的文件名后缀。

表 2-1 ARM 编译器可识别的文件名后缀

后缀	说明	使用说明
.c	C 源文件	相当于 --c90
.cpp .c++ .cxx .cc .CC	C++ 源文件	相当于 --cpp 编译器使用后缀 .cc 和 .CC 标识用于隐式包含的文件。请参阅 *《编译器参考指南》*中第5-15 页的*隐式包含*。
.d	相关性列表文件	.d 是使用 --md 选项输出的文件的缺省输出文件名后缀。
.h	C 或 C++ 头文件	--cpp --arm
.o .obj	ELF 格式的 ARM、 Thumb 或混合 ARM 和 Thumb 的对象文件。	
.s	ARM、 Thumb 或混合 ARM 和 Thumb 的汇编语言源文件。	对于输入文件列表中后缀为 .s 的文件，编译器调用汇编器 armasm 对文件进行汇编。 .s 是使用选项 -S 或 --asm 输出的文件的缺省输出文件名后缀。
.lst	错误和警报列表文件	.lst 是使用 --list 选项输出的文件的缺省输出文件名后缀。
.pch	预编译的头文件	.pch 是使用 --pch 选项输出的文件的缺省输出文件名后缀。
.txt	文本文件	.txt 是结合使用 -S 或 --asm 选项和 --interleave 选项输出的文件的缺省输出文件名后缀。

2.2.1 可移植性

为了确保主机之间的可移植性，请遵循以下准则：

- 确保文件名不包含空格。如果必须使用包含空格的路径名或文件名，请将路径和文件名置于双引号 (") 或单引号 (') 中。
- 使嵌入式路径名为相对路径而非绝对路径。
- 在嵌入式路径名中使用正斜杠 (/)，而不是反斜杠 (\)。

2.2.2 输出文件

缺省情况下，由 ARM 编译器创建的输出文件位于当前目录中。对象文件采用 *ARM 可执行和链接格式* (ELF) 进行编写。ELF 文档位于 *install_directory*\Documentation\Specifications\。

2.3 头文件

有多个因素会影响 ARM 编译器搜索 #include 头文件和源文件的方式。这些因素包括：

- 环境变量 RVCT40INC 的值
- -I 和 -J 编译器选项
- --kandr_include 和 --sys_include 编译器选项
- 文件名是绝对文件名还是相对文件名
- 文件名是在尖括号还是双引号之内。

请参阅：

- 《编译器参考指南》中第2-69 页的*-Idir[,dir,...]*
- 《编译器参考指南》中第2-76 页的*-Jdir[,dir,...]*
- 《编译器参考指南》中第2-76 页的*--kandr_include*
- 《编译器参考指南》中第2-118 页的*--sys_include*
- 《编译器参考指南》中第2-2 页的*命令行选项*。

2.3.1 当前位置

缺省情况下， ARM 编译器使用 Berkeley UNIX 搜索规则，因此，将相对于当前位置搜索源文件和 #include 头文件。这是包含编译器当前正在处理的源文件或头文件的目录。

当相对于搜索路径的某个元素找到文件后，包含该文件的目录变成新的当前位置。编译器处理完该文件后，会存储先前的当前位置。每一时刻都有一个当前位置堆栈与嵌套的 #include 指令堆栈对应。例如，如果当前位置是包含目录 ...\include，且编译器要搜索包含文件 sys\defs.h，则将会查找 ...\include\sys\defs.h （如果存在）。

当编译器开始处理 defs.h 时，当前位置变成 ...\include\sys。对于包含在未使用绝对路径名指定的 defs.h 中的任何文件，将相对于 ...\include\sys 进行搜索。

只有在编译器处理完 defs.h 后，才恢复原来的当前位置 ...\include。

通过使用编译器选项 --kandr_include，可以禁止当前位置形成堆栈。该选项允许编译器使用 Kernighan 和 Ritchie 最初在 《C 编程语言》中介绍的搜索规则。使用该规则时，将相对于包含要编译的源文件的目录搜索每个非根用户 #include。请参阅 《编译器参考指南》中第2-76 页的*--kandr_include*。

2.3.2 RVCT40INC 环境变量

RVCT40INC 环境变量指向 RVCT 附带的包含的头文件和源文件的位置。不要更改此环境变量。如果要包含其他位置的文件，请根据需要使用 -I 和 -J 命令行选项。

在编译过程中，搜索到由 -I 选项指定的目录后，将立即搜索由 RVCT40INC 指定的目录。如果使用 -J 选项，则忽略 RVCT40INC。

2.3.3 搜索路径

表 2-2 列出了在编译器搜索包含的头文件和源文件时，各命令行选项如何影响编译器所使用的搜索路径。

表 2-2 包含文件搜索路径

编译器选项	<include> 搜索顺序	"include" 搜索顺序
既不使用 -I 也不使用 -J	RVCT40INC*dirs*	*CP*、RVCT40INC*dirs*
-I	RVCT40INC*dirs*, I*dirs*	*CP*、I*dirs*、RVCT40INC*dirs*
-J	J*dirs*	*CP* 和 J*dirs*
同时使用 -I 和 -J	J*dirs*, I*dirs*	*CP*、I*dirs*、J*dirs*
--sys_include	无影响	从搜索路径中删除 *CP*
--kandr_include	无影响	使用 Kernighan 和 Ritchie 搜索规则

在表 2-2 中：

RVCT40INC*dirs*

由 RVCT40INC 环境变量 （如果设置）指定的目录列表。

CP 当前位置。

I*dirs* and J*dirs*

由 -I*dirs* 和 -J*dirs* 编译器选项指定的目录。

2.3.4 TMP 和 TMPDIR 环境变量

在 Windows 平台上，使用环境变量 TMP 指定供临时文件使用的目录。

在 Red Hat Linux 平台上，使用环境变量 TMPDIR 指定供临时文件使用的目录。如果未设置 TMPDIR，将使用缺省的临时目录，通常使用 /tmp 或 /var/tmp。

2.4 预编译的头文件

编译源文件时，也编译包含的头文件。如果一个头文件包含在多个源文件中，则编译每个源文件时都会重新编译它。有时，包含的头文件会生成许多行代码，但包含它的主源文件却相对较小。因此，经常需要对一组头文件进行预编译以避免重复编译。这些文件称为*预编译头* (PCH) 文件。

缺省情况下，在创建 PCH 文件时，编译器：

- 接受主源文件名，并用 .pch 取代后缀
- 在与主源文件相同的目录中创建文件。

——注意——

在一次编译中指定多个源文件时，不支持 PCH 处理。如果请求 PCH 处理并且指定多个主源文件，则编译器会发出一条错误消息并中止编译。

——注意——

不要假定当某个 PCH 文件可用时，编译器会使用该文件。在某些情况下，系统配置问题（例如 RHE3 和 Vista 上的地址空间随机化）意味着编译器有时可能无法使用 PCH 文件。

ARM 编译器可自动预编译头文件，也允许您控制预编译。请参阅：

- *自动 PCH 处理*
- 第2-20 页的*手动 PCH 处理*
- 第2-21 页的*控制 PCH 处理期间的消息输出*
- 第2-21 页的*性能问题*

2.4.1 自动 PCH 处理

使用 --pch 命令行选项时，将启用自动 PCH 处理。这意味着编译器自动查找合格的 PCH 文件，并在找到后读取该文件。否则，编译器将创建在随后编译中使用的文件。

当编译器创建 PCH 文件时，会接受主源文件名，并用 .pch 取代后缀。除非指定 --pch_dir 选项，否则在主源文件的目录中创建 PCH 文件。

请参阅第2-10 页的*命令行选项排序*。

头文件终止点

PCH 文件包含*头文件终止点*之前的所有代码的快照。通常，头文件终止点是不属于预处理指令的主源文件中的第一个标记。在下例中，头文件终止点是 int，并且 PCH 文件包含反映同时含有 xxx.h 和 yyy.h 的快照：

```
#include "xxx.h"
#include "yyy.h"
int i;
```

注意

可以使用 #pragma hdrstop 手动指定头文件终止点。它必须置于不属于预处理指令的第一个标记之前。在本例中，将其置于 int 之前。请参阅第2-20 页的*控制 PCH 处理*。

影响 PCH 文件生成的条件

如果第一个非预处理器标记或 #pragma hdrstop 出现在 #if 块中，则头文件终止点是最外层的 #if。例如：

```
#include "xxx.h"
#ifndef YYY_H
#define YYY_H 1
#include "yyy.h"
#endif
#if TEST
int i;
#endif
```

在本例中，不属于预处理指令的第一个标记是 int，但是包含它的 #if 块的开头才是头文件终止点。PCH 文件反映是否包含 xxx.h，在某些条件下还反映 YYY_H 的定义以及是否包含 yyy.h。它不包含由 #if TEST 生成的状态。

只有当头文件终止点和它 （主要指头文件）前面的代码满足以下要求时，才生成 PCH 文件：

- 头文件终止点必须出现在文件范围内。它不能处于由头文件设定的未闭合范围内。例如，在这种情况下不创建 PCH 文件：

```
// xxx.h
class A
{
    // xxx.c
    #include "xxx.h"
    int i;
};
```

- 头文件终止点不能位于开始于头文件的声明中。同样，在 C++ 中它不能是链接规范的声明列表的组成部分。例如，如果属于以下情况，头文件终止点是 int，但是因为它不是新声明的开头，所以不创建 PCH 文件：

```
// yyy.h
static
// yyy.c
#include "yyy.h"
int i;
```

- 头文件终止点不能位于开始于头文件的 #if 块或 #define 中。
- 头文件终止点之前的处理不能产生任何错误。

注意

重新使用 PCH 文件时，不重现警告和其他诊断。

- 不必出现对预定义宏 __DATE__ 或 __TIME__ 的引用。
- 不必出现 #line 预处理指令的实例。
- 不能出现 #pragma no_pch。
- 头文件终止点之前的代码必须引入足够多的声明，以证明与预编译头文件相关联的开销合理。

可将多个 PCH 文件应用于指定编译。如果是这样，则使用最大的文件，即代表主源文件的最多预处理指令的文件。例如，主源文件可能开始于：

```
#include "xxx.h"
#include "yyy.h"
#include "zzz.h"
```

如果有一个 PCH 文件用于 xxx.h，第二个 PCH 文件用于 xxx.h 和 yyy.h，假定这两个文件都应用于当前编译，则选择后者。另外，读入用于前两个头文件的 PCH 文件并编译完第三个头文件后，可为这三个头文件新建一个 PCH 文件。

在自动 PCH 处理模式中，以下情况下编译器会指示过时的 PCH 文件并将其删除。

- PCH 文件基于至少一个过时的头文件，否则可应用于当前编译
- PCH 文件与正在编译的源文件有相同的基本名称 （例如，xxx.pch 和 xxx.c），但是不适用于当前编译 （例如，因为使用了不同的命令行选项）。

上面介绍的是一些常见情况。您必须根据需要删除其他 PCH 文件。

2.4.2 手动 PCH 处理

可以指定 PCH 文件的文件名和位置，以及头文件中要进行 PCH 处理的部分。

指定 PCH 文件名和位置

可以使用以下命令行选项指定 PCH 文件的文件名和位置：

- `--create_pch=filename`
- `--use_pch=filename`
- `--pch_dir=directory`

如果将 `--create_pch` 或 `--use_pch` 与 `--pch_dir` 选项结合使用，则除非文件名是绝对路径名，否则指定的文件名将附加至目录名。

排序 PCH 命令行选项

编译器不能在同一命令行中同时使用这三个选项。如果指定了这些选项中的多个选项，则遵循以下规则：

- `--use_pch` 优先于 `--pch`
- `--create_pch` 优先于所有其他的 PCH 选项。

自动 PCH 处理的大部分功能适用于这些模式中的一个或另一个。例如，以同样的方式确定头文件终止点和 PCH 文件适用性。

控制 PCH 处理

可以使用以下编译指示指定头文件中进行 PCH 处理的部分：

- 在不属于预处理指令的第一个标记之前，使用主源文件中的 `#pragma hdrstop` 指令插入手动头文件终止点。

  这样可以指定要进行预编译的头文件集的结束位置。例如，

  ```
  #include "xxx.h"
  #include "yyy.h"
  #pragma hdrstop
  #include "zzz.h"
  ```

  在本例中，PCH 文件包含 `xxx.h` 和 `yyy.h` 的处理状态，但是没有 `zzz.h` 的处理状态。判定 `#pragma hdrstop` 后面的信息不能证明创建另一个 PCH 文件合理时，它很有用。

- 使用 `#pragma no_pch` 指令禁止对源文件进行 PCH 处理。

注意

即使要使用自动 PCH 处理，也可使用这些编译指示。

请参阅第4-13 页的*编译指示*。

2.4.3 控制 PCH 处理期间的消息输出

当编译器创建或使用 PCH 文件时，会显示以下消息：

```
test.c: creating precompiled header file test.pch
```

可以使用命令行选项 `--no_pch_messages` 禁止显示该消息。

当使用 `--pch_verbose` 选项时，对于考虑到但不能使用的每个 PCH 文件，编译器都会显示一条消息，同时提供不能使用的原因。

2.4.4 性能问题

通常，创建和读取 PCH 文件的开销不大，即使头文件相当大或者创建的 PCH 文件未使用，也是如此。如果使用到 PCH 文件，会显著地减少编译时间。但是，PCH 文件的大小从 250KB 到几兆字节甚至更多，因此您可能并不想创建许多的 PCH 文件。

PCH 处理有时并不适用，例如，对于具有预处理指令的非统一初始序列的任意文件组。

当多个源文件可共享同一 PCH 文件时可显示出 PCH 处理的优势。共享越多，磁盘空间消耗越少。通过共享可将大型 PCH 文件的缺点减至最小，并保持明显缩短编译时间的优势。

因此，要充分利用头文件预编译的优势，就必须对源文件的 `#include` 节重新排序，或对常用头文件中的 `#include` 指令进行分组。

不同环境和不同项目可能有不同的要求。但值得注意的是，要充分利用 PCH 支持，可能需要进行一些试验，并对源代码稍作更改。

2.5 指定目标处理器或体系结构

RVCT 提供对所有 ARM 体系结构 （ARMv4 及更高版本）的支持，包括 ARM NEON™ 技术。ARMv4 之前的所有体系结构名称现已过时，不再受支持。

通过指定目标处理器或体系结构，可使编译器利用所选处理器或体系结构特有的附加功能。使用 `--cpu` 和 `--fpu` 选项可以启用这些功能。

此外，通过使用 `--arm` 或 `--thumb` 选项，可以指定启动指令集。

请参阅：

- *NEON 技术*
- 第5-3 页的*选择目标 CPU*
- *《开发指南》*中第 5 章 *交互操作 ARM 和 Thumb*。

另请参阅 *《编译器参考指南》*中以下内容：

- 第2-8 页的*--arm*
- 第2-30 页的*--cpu=list*
- 第2-30 页的*--cpu=name*
- 第2-61 页的*--fpu=list*
- 第2-61 页的*--fpu=name*
- 第2-119 页的*--thumb*

2.5.1 NEON 技术

ARM 高级单指令多数据 (SIMD) 扩展亦称 NEON 技术，它是一种由 ARM 开发的 64/128 位混合 SIMD 体系结构，可以提升多媒体和信号处理应用程序的性能。NEON 作为处理器的一部分来实现，但是它拥有自己的执行管道，以及有别于 ARM 的寄存器组。关键功能包括对齐和未对齐数据访问，支持整型定点和单精度浮点数据类型、与 ARM 核心的紧密耦合，以及具有多个视图的大型寄存器文件。NEON 指令在 ARM 和 Thumb-2 中都可用。

ARM 编译器支持配备有 NEON 单元的 Cortex™ 处理器。要生成 NEON 指令，必须在命令行中指定采用 NEON 技术的 Cortex 处理器，例如 `--cpu=Cortex-A8`。ARMv7 之前的体系结构不提供 NEON 支持。

请参阅 *《编译器参考指南》*中附录 E *使用 NEON 支持*。

2.6 指定过程调用标准 (AAPCS)

《ARM 体系结构的过程调用标准》 (*Procedure Call Standard for the ARM Architecture*) (AAPCS) 是 《ARM 体系结构的基础标准应用程序二进制接口》 (*Base Standard Application Binary Interface for the ARM Architecture*) (BSABI) 规范的组成部分。遵循 AAPCS 编写代码可以确保分别编译和汇编的模块能够协同工作。

请参阅：

- *交互操作限定符*
- *位置无关限定符*
- 第2-22 页的*指定目标处理器或体系结构*
- 《ARM 体系结构的过程调用标准》 (*Procedure Call Standard for the ARM Architecture*) 规范 `aapcs.pdf`，该文件位于 *`install_directory`*`\Documentation\Specifications\....`

2.6.1 交互操作限定符

这些 `--apcs` 限定符控制交互操作。

请参阅：

- *《编译器参考指南》*中第2-4 页的*`--apcs=qualifer...qualifier`*
- *《开发指南》*中第 5 章 *交互操作 ARM 和 Thumb*
- *《链接器用户指南》*中第 3 章 *使用基本链接器功能*。

2.6.2 位置无关限定符

这些 `--apcs` 限定符控制位置无关性。它们还会影响可重入且线程安全的代码的创建。

请参阅：

- *《编译器参考指南》*中第2-4 页的*`--apcs=qualifer...qualifier`*
- 第2-24 页的*位置无关代码和数据的限制*
- *《库指南》*中第2-5 页的*编写可重入且线程安全的代码*
- *《链接器参考指南》*中第 4 章 *BPABI 和 SysV 共享库和可执行文件*。

位置无关代码和数据的限制

使用 /ropi、 /rwpi 或 /fpic 编译代码时存在一些限制。主要的限制如下：

- 编译 C++ 时，不支持使用 --apcs /ropi。只能使用 /ropi 编译 C++ 的 C 子集。
- 一些合法的 C 结构在使用 --apcs=/ropi 或 --apcs=/rwpi 编译时不起作用，例如：

  ```
  int i;               // rw
  int *p1 = &i;        // this static initialization does not work
                       // with --apcs=/rwpi --no_lower_rwpi
  extern const int ci; // ro
  const int *p2 = &ci; // this static initialization does not work
                       // with --apcs=/ropi
  ```

  但是，要使这些静态初始化有效，请使用 --lower_rwpi 和 --lower_ropi 选项。

  要编译此代码，请键入：

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

  因为 --lower_rwpi 是缺省选项，所以不必另行指定。
- 编译 C++ 时，不支持使用 --apcs=/fpic。此时，将把虚拟表函数和 typeinfo 放在读写区域中，以便相对于 PC 的位置对其进行访问。
- 如果使用 --apcs=/fpic，则编译器只导出标有 __declspec(dllexport) 的函数和数据。
- 如果使用 --no_hide_all，则编译器对不使用 __declspec(dll*) 的所有 extern 变量和函数使用 STV_DEFAULT 可见性。编译器禁用具有 STV_DEFAULT 可见性的函数的自动内联。

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

有关 __declspec 关键字的详细信息，请参阅 *《编译器参考指南》*中第4-24 页的 *__declspec 属性*。

有关符号可见性的详细信息，请参阅 *《链接器参考指南》*中第4-4 页的*符号可见性*。

2.7 使用链接器反馈

链接器提供的反馈功能可以：

- 高效删除未使用的函数
- 减少进行交互操作所需的编译。

2.7.1 删除未使用的函数

在以下情况下，可能出现未使用的函数代码：

- 源代码中有不再使用的旧函数时。可以使用链接器反馈从最终映像中自动删除未使用的对象代码，而不用从源代码中手动删除未使用的函数代码。
- 函数处于内联状态时。如果内联函数未声明为 **static**，外部函数代码仍在对象文件中，但不再调用该代码。

从对象文件中删除未使用的函数：

1. 编译源代码。
2. 使用链接器选项 `--feedback=`*filename* 创建反馈文件。缺省情况下，生成的反馈类型用于删除未使用的函数。
3. 使用编译器选项 `--feedback=`*filename* 将反馈文件发送给编译器。

编译器使用链接器生成的反馈文件，以链接器可以随后丢弃未使用函数的方式编译源代码。

注意

若要最大程度地利用链接器反馈，请至少执行两次完全编译和链接。通常，使用上次生成的反馈进行单次编译和链接就可受益匪浅。

即便反馈文件不存在，也能指定 `--feedback=`*filename* 选项。这样，无论反馈文件存在与否，都能使用同一个构建文件或 make 文件，例如：

```
armcc -c --feedback=unused.txt test.c -o test.o
armlink --feedback=unused.txt test.o -o test.axf
```

首次构建应用程序时会正常编译，但编译器会发出警告，指示因指定的反馈文件不存在而无法读取。然后，链接命令创建反馈文件并生成映像。每个后续编译步骤都使用前一链接步骤的反馈文件来删除所标识的所有未使用函数。

请参阅：

- 《编译器参考指南》中第 2-56 页的 *--feedback=filename*
- 《链接器参考指南》中第 2-22 页的 *--feedback_type=type*
- 《链接器用户指南》中第 3-13 页的*反馈*。

2.7.2 减少进行交互操作所需的编译

注意

减少交互操作编译只适用于 ARMv4T 体系结构。ARMv5T 及更高版本处理器的交互操作不会影响性能。

链接器检测从 Thumb 状态调用 ARM 函数或从 ARM 状态调用 Thumb 函数时的情况。通过使用链接器反馈，可以避免编译不会在交互操作上下文中使用的交互操作函数。

减少交互操作编译：

1. 编译源代码。
2. 使用链接器选项 `--feedback=`*filename* 和 `--feedback_type=iw` 创建一个反馈文件，用于报告需要交互操作支持的函数。
3. 使用编译器选项 `--feedback=`*filename* 将反馈文件发送给编译器。

编译器使用链接器生成的反馈文件编译源代码，并且编译器不编译不会在交互操作上下文中使用的函数。

注意

请务必确保紧接在使用链接器反馈文件之前执行完全完整生成。这样可最大程度地降低用过时源代码生成反馈文件的可能性。

请参阅：

- 《编译器参考指南》中第 2-56 页的 *--feedback=filename*
- 《链接器参考指南》中第 2-22 页的 *--feedback_type=type*
- 《链接器用户指南》中第 3-13 页的*反馈*。

2.8 添加符号版本

编译器和链接器支持 GNU 扩展符号版本模型。

若要使用 C 或 C++ 代码创建带符号版本的函数，必须使用汇编器标签 GNU 扩展将函数符号重命名为一个符号，对于 *function* 的缺省 *ver*，该符号的名称为 *function*@@*ver*，对于 *function* 的非缺省 *ver*，该符号的名称为 *function*@*ver*。

例如，定义缺省版本：

```
int new_function(void) __asm__("versioned_fun@@ver2");
int new_function(void)
{
    return 2;
}
```

定义非缺省版本：

```
int old_function(void) __asm__("versioned_fun@ver1");
int old_function(void)
{
    return 1;
}
```

请参阅：

- *《编译器参考指南》*中第3-20 页的*汇编器标签*
- *《链接器参考指南》*中第4-15 页的*符号版本控制*。

第 3 章
使用 NEON 向量化编译器

本章介绍 NEON™ 单元，并说明如何利用自动向量化功能。本章分为以下几节：

3.1 NEON 单元

NEON 单元提供 32 个向量寄存器，每个寄存器可保存 16 字节的信息。然后，这些 16 字节寄存器可以在 NEON 单元中进行并行运算。例如，在一个向量相加指令中，您可以将 8 个 16 位整数加上另外 8 个 16 位整数，以得到 8 个 16 位的结果。

NEON 单元支持 8 位、16 位、32 位整数运算和部分 64 位运算，以及 32 位浮点运算。

注意

浮点代码的向量化并不总是自动进行。例如，需要重新关联的循环只在使用 `--fpmode fast` 编译时进行向量化。如果使用 `--fpmode fast` 编译，则编译器可以执行某些可能影响结果的转换。（请参阅 *《编译器参考指南》* 中第2-59 页的*`--fpmode=model`*。

NEON 单元被归类为向量 SIMD 单元，可使用一条指令在一个向量寄存器中对多个元素进行运算。

例如，数组 A 是一个有 8 个元素的 16 位整数数组。

表 3-1 数组 A

1	2	3	4	5	6	7	8

数组 B 也有这 8 个元素：

表 3-2 数组 B

80	70	60	50	40	30	20	10

若要将这些数组相加，提取每个向量放入向量寄存器中，并使用一个向量 SIMD 指令来得出结果。

表 3-3 结果

81	72	63	54	45	36	27	18

3.2 编写用于 NEON 的代码

本节概要介绍可用于编写 NEON 单元代码的方式。可以通过几种方式来获得在 NEON 上运行的代码：

- 使用汇编语言、使用 C 中的嵌入式汇编语言以及直接使用 NEON 指令来编写代码。
- 使用具有 NEON C 语言扩展的 C 或 C++ 编写代码。
- 调用已优化为使用 NEON 指令的库例程。
- 使用自动向量化功能来获得针对 NEON 向量化的循环。

3.2.1 NEON C 扩展

NEON C 扩展是由 ARM 定义的一组新的数据类型和内在函数，用于从 C 访问 NEON 单元。大部分向量函数直接映射到 NEON 单元中提供的向量指令，并且由 NEON 增强的 ARM C 编译器进行内联编译。通过使用这些扩展，性能可以达到 C 语言的级别，这可以与通过汇编语言编码达到的性能相媲美。

请参阅 《编译器参考指南》中附录 E *使用 NEON 支持*。

3.2.2 自动向量化

通过使用可向量化的循环进行编码 （而不是使用显式 NEON 指令编写代码），可以保留代码在处理器之间的可移植性。这样可以轻松达到与手写编码向量化类似的性能级别。

示例 3-1 是在 Cortex-A8 处理器上调用自动向量化的命令行选项的示例。

示例 3-1 自动向量化

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

——注意——

您还可以使用 `-O2 -Otime` 进行编译，但是这样不会得到最佳代码性能。

3.2.3 性能目标

大部分应用程序需要优化，以获得最佳向量化性能。开销始终会有一些，因此理论上是不可能达到最佳性能的。例如， NEON 单元可以同时处理四个单精度浮点数。这意味着，浮点应用程序的最佳理论性能是原始标量非向量化代码的四倍。在典型的开销下，整个浮点应用程序的合理目标是比标量代码的性能提高 50%。对于不可完全向量化的大型应用程序，比标量代码的性能提高 25% 是比较合理的目标，尽管这在很大程度上取决于具体的应用程序。

请参阅第3-7 页的*改善性能*。

3.3 使用自动向量化

本节概要介绍自动向量化，还介绍影响向量化过程和生成代码性能的因素。

3.3.1 自动向量化概述

自动向量化涉及对您代码中循环的高级分析。这是将大部分典型代码映射到 NEON 单元功能的最有效的方法。对于大部分代码，小规模使用算法相关的并行处理所得到的收获与自动分析的开销相比是微不足道的。因此， NEON 单元专门针对基于循环的并行处理。

向量化的执行方式确保优化代码得出的结果与非向量化代码相同。在某些情况下，不执行循环的向量化，以避免可能出现错误结果。这可能会导致代码不是最优，您可能需要手动优化代码，以使其更加适合自动向量化。请参阅第3-7 页的*改善性能*。

3.3.2 向量化概念

本节介绍在考虑代码向量化时的一些常用概念。

数据引用

代码中的数据引用可以分为以下三种类型：

标量 是在所有循环迭代过程中均保持不变的单一位置。

嶕堷 是随着每次循环而增加一个常数的整数。

向量 连续元素间具有常数跨度的内存位置范围。

示例 3-2 显示了循环中变量的分类：

`i,j` 索引变量
`a,b` 向量
`x` 标量

示例 3-2 可向量化循环的类别

```
float *a, *b;
int i, j, n;
...
for (i = 0; i < n; i++)
{
```

```
    *(a+j) = x +b[i];
    j += 2;
};
```

若要进行向量化，编译器必须使用向量访问模式标识变量。还必须确保在循环的不同迭代之间不存在任何数据相关性。

跨度模式和数据访问

循环中数据访问的跨度模式是一种对顺序循环迭代间数据元素的访问模式。例如，线性访问数组中每个元素的循环的跨度为 1。另一个例子是，如果循环使用元素间的常数偏移量对数组进行访问，则该循环使用常数跨度。在第3-5 页的示例 3-2 中，使用跨度 1 访问 b，而使用跨度 2 访问 a。

3.3.3 影响向量化性能的因素

自动向量化过程和生成代码的性能受以下因素影响：

组织循环的方式

要获得最佳性能，循环嵌套中最内层的循环必须以跨度 1 访问数组。

组织数据的方式

数据类型指示可以在 NEON 寄存器中保存的数据元素的数量，以及可并行执行的运算数量。

循环的迭代计数

通常较长的迭代计数比较好，因为循环的开销可以分摊在更多迭代中。较小的迭代计数 （如两个或三个元素）可以通过非向量指令更快地进行处理。

数组的数据类型

例如，在使用双精度浮点数组时， NEON 不会改善性能。

内存层次结构的使用

相对而言，大部分当前的处理器在内存带宽和处理器容量之间都是不平衡的。例如，对从主内存检索的大型数据集执行相对较少的算术运算时会受到系统内存带宽的限制。

3.3.4 改善性能

大部分应用程序要求程序员执行一些优化，以获得最佳 NEON 结果。本节介绍不同类型的循环。它解释了向量化如何对某些循环起作用，而对其他循环不起作用。它还解释了可以如何修改代码以使向量化代码获得最佳性能。

一般性能问题

使用命令行选项 -O3 和 -Otime 可确保除了向量化的好处之外，代码还可获得显著的性能优势。

优化性能时，您必须考虑到高级算法结构、数据元素大小、数组配置、严格的迭代循环、缩减运算和数据相关性问题。性能优化要求了解程序中最花费时间之处。若要获得最佳性能优势，可能需要使用符合实际条件的性能分析和代码基准测试。

先前对代码进行的任何手动优化 （例如，对源代码或复杂数组访问进行手动循环展开）常常会妨碍自动向量化。若要获得最佳结果，最好的办法是使用简单循环编写代码，从而使编译器执行全部优化。对于手动优化的旧代码，根据原始算法使用简单循环重新编写关键部分可能会更容易。

请参阅 《编译器参考指南》中的以下内容：

- *《编译器参考指南》*中第2-127 页的*--vectorize*, *--no_vectorize*
- *《编译器参考指南》*中第2-94 页的*-Onum*
- *《编译器参考指南》*中第2-97 页的*-Otime*。

数据相关性

如果循环的一次迭代结果反馈到该循环的未来迭代，则这种循环存在数据相关性冲突。冲突的值可能是数组元素或标量，如累加求和。

包含数据相关性冲突的循环可能未完全优化。要检测与数组和/或指针有关的数据相关性，要求广泛分析在每个循环嵌套中使用的数组，并对在循环中使用和存储的数组，检查对其每个维度元素的访问的偏移量和跨度。如果在循环的不同迭代中存在数组使用和存储重叠的可能性，则会出现数据相关性问题。如果运算的向量顺序会更改结果，则循环无法安全地向量化。在这类情况下，编译器检测该问题，并使循环保持原始格式或执行循环的部分向量化。在您的代码中必须避免这种类型的数据相关性，以获得最佳性能。

在第3-8 页的示例 3-3 显示的循环中，对循环顶部的 a[i-2] 的引用与循环底部的存入 a[i] 相冲突。对此循环执行的向量化会得出不同的结果，因此该循环保持原始格式。

示例 3-3 不可向量化的数据相关性

```
float a[99], b[99], t;
int i;
for (i = 3; i < 99; i++)
{
    t = a[i-1] + a[i-2];
    b[i] = t + 3.0 + a[i];
    a[i] = sqrt(b[i]) - 5.0;
};
```

来自其他数组下标的信息用作相关性分析的一部分。示例 3-4 中的循环进行了向量化，这是因为对数组 a 引用的非向量下标永远不可能相等，因为 n 不等于 n+1，因此不会提供迭代间的反馈。对数组 a 的引用使用两个不同的数组，因此不会共享数据。

示例 3-4 可向量化的数据相关性

```
float a[99][99], b[99], c[99];
int i, n, m;
...
for (i = 1; i < m; i++) a[n][i] = a[n+1][i-1] * b[i] + c[i];
```

标量变量

在 NEON 循环中使用但未设置的标量变量，会复制到向量寄存器的每个位置和向量计算中使用的每个结果中。

设置后在循环中使用的标量*升级*为向量。这些变量通常保存循环中的临时标量值，现在需要保存临时向量值。在示例 3-5 中，x 是一个*已使用的*标量，y 是一个*升级的*标量。

示例 3-5 可向量化的循环

```
float a[99], b[99], x, y;
int i, n;
...
for (i = 0; i < n; i++)
{
```

```
    y = x + b[i];
    a[i] = y + 1/y;
};
```

使用后在循环中设置的标量称为*传递*标量。这些变量对于向量化是个问题，因为在一轮循环中计算的值将传递到下一轮循环。在 示例 3-6 中，x 是一个传递标量。

示例 3-6 不可向量化的循环

```
float a[99], b[99], x;
int i, n;
...
for (i = 0; i < n; i++)
{
    a[i] = x + b[i];
    x = a[i] + 1/x;
};
```

缩减运算

循环中一种特殊类别的标量用途是缩减运算。这种类别包括将值的向量缩减为标量结果。最常见的缩减是将向量的所有元素相加。其他缩减包括：两个向量的点积、向量最大值、向量最小值、所有向量元素的积和向量中最大或最小元素的位置。

示例 3-7 显示了标量积缩减，其中 x 是缩减标量。

示例 3-7 标量积缩减

```
float a[99], b[99], x;
int i, n;
...
for (i = 0; i < n; i++) x += a[i] * b[i];
```

缩减运算有必要向量化，因为它们经常出现。通常，通过创建部分缩减的向量，然后将其缩减为最终结果标量的方式来向量化缩减运算。

使用指针

在访问数组时，编译器通常可以验证内存访问没有重叠。在使用指针时，这不太可能，因而需要进行运行时测试或使用 **restrict**。

如果编译器能够确定包含指针的循环是安全的，则可以向量化该循环。循环中的数组引用和指针引用都要分析，以查看是否存在任何对内存的向量访问。在某些情况下，编译器创建运行时测试，并根据测试结果确定执行循环的向量版本还是标量版本。

通常，函数参数是作为指针传递的。如果向函数传递多个指针变量，则可能发生指向的内存重叠的情况。在运行时通常不会这样，但编译器始终遵循安全的方法，避免优化在赋值运算符左侧和右侧同时出现指针的循环。例如，考虑示例 3-8 中的函数。

示例 3-8 指针的条件向量化

```
void func (int *pa, int *pb, int x)
{
    int i;
    for (i = 0; i < 100; i++) *(pa + i) = *(pb + i) + x;
};
```

在此示例中，如果 pa 和 pb 在内存中的重叠会导致一次循环的结果反馈到后续循环，则该循环的向量化可能产生错误的结果。如果使用以下自变量调用函数，则向量化可能不明确：

```
int *a;

func (a, a-1);
```

编译器执行运行时测试，以查看是否出现指针别名。如果未出现指针别名，则执行代码的向量化版本。如果出现指针别名，则改为执行原始的非向量化代码。这会导致运行时效率和代码大小方面的少量开销。

在实际中，很少有函数参数引起的数据相关性。除了向量化的问题之外，传递重叠指针的程序还很难理解和调试。

请参阅 《编译器参考指南》中第3-7 页的*restrict*。在示例 3-8 中，向 pa 添加了 **restrict** 就不用进行运行时测试。

间接寻址

当值的向量访问数组时，发生间接寻址。如果要从内存获取数组，则该运算称为*集中*。如果数组要存储在内存中，则该运算称为*分散*。在示例 3-9 中，a 进行分散，b 进行聚合。

示例 3-9 不可向量化的间接寻址

```
float a[99], b[99];
int ia[99], ib[99], i, n, j;
...
for (i = 0; i < n; i++) a[ia[i]] = b[j + ib[i]];
```

间接寻址对 NEON 单元不可向量化，因为它只处理内存中连续存储的向量。如果循环中存在间接寻址和大量计算，则将间接寻址移到单独的非向量循环中可能会更加有效。这样可以使向量化计算更有效。

循环结构

若要从向量化中获取最佳性能，循环的整体结构是很重要的。通常，最好编写简单的循环，其迭代计数在循环开始时即固定，并且循环中不包含复杂的条件语句或条件退出。您可能需要重新编写循环，以改善代码的向量化性能。

从循环中退出

示例 3-10 也无法进行向量化，因为它在循环中包含退出。在这种情况下，如果可能，必须重新编写该循环，以使向量化成功执行。

示例 3-10 不可向量化的循环

```
int a[99], b[99], c[99], i, n;
...
for (i = 0; i < n; i++)
{
    a[i] = b[i] + c[i];
    if (a[i] > 5) break;
};
```

循环迭代计数

循环在开始时即必须有固定的迭代计数。示例 3-11 的迭代计数为 n，并且该计数不随循环的进行而变化。

示例 3-11 可向量化的循环

```
int a[99], b[99], c[99],i, n;
...
for (i = 0; i < n; i++) a[i] = b[i] + c[i];
```

示例 3-12 没有固定的迭代计数，因而无法自动进行向量化。

示例 3-12 不可向量化的循环

```
int a[99], b[99], c[99], i, n;
...
while (i < n)
{
    a[i] = b[i] + c[i];
    i += a[i];
};
```

NEON 单元可以对分为 2、4、8 或 16 组的元素进行运算。如果迭代计数在循环开始时已知，则编译器可能会增加一个运行时测试，以检查迭代计数是否不是可用于 NEON 寄存器中相应数据类型的向量线的倍数。这会增加代码的大小，因为生成了执行任何附加循环迭代的附加非向量化代码。

如果您知道迭代计数是 NEON 所支持的迭代计数之一，则可以向编译器指示此计数。执行此操作最有效的方式是在调用方将迭代数除以四，在要向量化的函数中将迭代数乘以四。如果您无法修改所有的调用函数，则可以将适当的表达式用于循环限制测试，以指示循环迭代是一个合适的倍数。例如，若要指示循环迭代次数是四的倍数，可使用：

```
for(i = 0; i < (n >> 2 << 2); i++)
```

或：

```
for(i = 0; i < (n & ~3); i++)
```

这会缩减生成代码的大小，并改善性能。

对于向量化，编写循环以将结构的所有部分作为整体使用是很重要的。结构的每个部分都需要在同一个循环中进行访问。

示例 3-13 不可向量化的循环

```
for (...) { buffer[i].a = ....; }
for (...) { buffer[i].b = ....; }
for (...) { buffer[i].c = ....; }
```

示例 3-14 可向量化的循环

```
for (...)
{
    buffer[i].a = ....;
    buffer[i].b = ....;
    buffer[i].c = ....;
}
```

函数调用和内联

在循环内调用非内联函数会禁止向量化。

通常将复杂的运算拆分为多个函数，可使运算更加明了。为了在向量化时将这些函数考虑在内，必须使用 `__inline` 或 `__forceinline` 关键字对其进行标记。然后，这些函数扩展为内联，以进行向量化。请参阅 *《编译器参考指南》*中第4-9 页的*__inline* 和 第4-6 页的*__forceinline*。

条件语句

若要进行有效的向量化，循环中必须包含大部分赋值语句，并限制使用 **if** 和 **switch** 语句。

循环迭代之间不会变化的简单条件称为循环不变量。编译器可在循环之前将这些不变量移走，以便不在每个循环迭代中执行它们。通过计算向量模式下的所有路径并合并结果来对更复杂的条件运算进行向量化。如果要有条件地执行大量计算，则会浪费大量时间。

第3-14 页的示例 3-15 演示一种可接受的条件语句的用法。

示例 3-15 可向量化的条件

```
float a[99], b[99], c[i];
int i, n;
...
for (i = 0; i < n; i++)
{
    if (c[i] > 0) a[i] = b[i] - 5.0;
    else a[i] = b[i] * 2.0;
};
```

结构

NEON 结构加载要求结构的所有成员具有相同的长度。因此，编译器不会尝试对示例 3-16 中所示的代码使用向量加载。

示例 3-16 由不一致的数据类型所导致的不可向量化的代码

```
struct foo
{
    short a;
    int b;
    short c;
} n[10];
```

通过在整个结构中使用相同数据类型来重新编写示例 3-16 中的代码，则可以进行向量化。例如，如果 b 要作为整数数据类型，则考虑将 a 和 c 替换为整数数据类型。

在结构中填充会禁止向量化。在示例 3-17 中，对齐每个 a 变量没有益处，因为 NEON 单元加载未对齐结构不会影响性能。

示例 3-17 由对齐填充所导致的不可向量化的代码

```
struct aligned_data
{
    char a;
    char b;
    char c;
    char not_used;
} n[10];
```

通过删除 not_used 填充重新编写第3-14 页的示例 3-17 中的代码，则可以进行向量化。

通过优化源代码来改善性能的示例

编译器可提供诊断信息来指示在何处成功应用了向量化优化，在何处应用向量化失败。请参阅 *《编译器参考指南》*中第2-47 页的 *--diag_suppress=optimizations* 和第2-48 页的 *--diag_warning=optimizations*。

示例 3-18 是两个对数组实现简单求和运算的函数。此代码未向量化。

示例 3-18 不可向量化的代码

```
int addition(int a, int b)
{
    return a + b;
}
void add_int(int *pa, int *pb, unsigned int n, int x)
{
    unsigned int i;
    for(i = 0; i < n; i++) *(pa + i) = addition(*(pb + i),x);
}
```

使用 --diag_warnings=optimization 选项将对 addition() 函数产生优化警告消息。

向 addition() 的定义添加 __inline 限定符可使此代码向量化，但仍未优化。再次使用 --diag_warnings=optimization 选项会产生优化警告消息，指示循环得以向量化，但是可能存在潜在的指针别名问题。

编译器必须生成针对别名的运行时测试，并输出代码的向量化副本和标量副本。示例 3-19 演示了当您知道指针没有别名时，如何使用 restrict 关键字进行改善。

示例 3-19 使用 restrict 改善向量化性能

```
__inline int addition(int a, int b)
{
    return a + b;
}
void add_int(int * __restrict pa, int * __restrict pb, unsigned int n, int x)
{
```

```
    unsigned int i;
    for(i = 0; i < n; i++) *(pa + i) = addition(*(pb + i),x);
}
```

最后可进行的改善是循环迭代的次数。在第3-15 页的示例 3-19 中，迭代的次数是不固定的，并且可能不是正好适合 NEON 寄存器的倍数。这意味着，编译器必须进行测试，以使用非向量化的代码执行剩余的迭代。如果知道迭代计数受 NEON 支持，则可以提示编译器。示例 3-20 演示为获得最佳向量化性能可以进行的最终改进。

示例 3-20 为获得最佳向量化性能而优化的代码

```
__inline int addition(int a, int b)
{
    return a + b;
}
void add_int(int * __restrict pa, int * __restrict pb, unsigned int n, int x)
{
    unsigned int i;
    for(i = 0; i < (n & ~3); i++) *(pa + i) = addition(*(pb + i),x);
    /* n is a multiple of 4 */
}
```

使用 __promise 改善向量化性能

__promise(*expr*) 内在函数向编译器保证给定的表达式是非零的。这将允许编译器基于所做保证无需优化冗余代码，从而改善向量化性能。

示例 3-21 的反汇编输出演示了因使用 __promise 而带来的差异，通过删除标量固定循环而将反汇编代码减少为一个简单的向量化循环。

示例 3-21 使用 __promise(*expr*) 改善向量化代码的性能

```
void f(int *x, int n)
{
    int i;
    __promise((n > 0) && ((n&7)==0));
    for (i=0; i<n;i++) x[i]++;
}
```

示例 3-20 是一个因使用 __promise() 而受益的类似示例。

3.4 示例

以下是可向量化的代码的示例。请参阅示例 3-22 和第3-19 页的示例 3-23。

示例 3-22 向量化代码

```
/*
 * Vectorizable example code.
 * Copyright 2006 ARM Limited. All rights reserved.
 *
 * Includes embedded assembly to initialize cpu; link using '--entry=init_cpu'.
 *
 * Build using:
 *   armcc --vectorize -c vector_example.c --cpu Cortex-A8 -Otime
 *   armlink -o vector_example.axf vector_example.o --entry=init_cpu
 */
#include <stdio.h>
void fir(short *__restrict y, const short *x, const short *h, int n_out, int n_coefs)
{
    int n;
    for (n = 0; n < n_out; n++)
    {
        int k, sum = 0;
        for (k = 0; k < n_coefs; k++)
        {
            sum += h[k] * x[n - n_coefs + 1 + k];
        }
        y[n] = ((sum>>15) + 1) >> 1;
    }
}
int main()
{
    static const short x[128] =
    {
        0x0000, 0x0647, 0x0c8b, 0x12c8, 0x18f8, 0x1f19, 0x2528, 0x2b1f,
        0x30fb, 0x36ba, 0x3c56, 0x41ce, 0x471c, 0x4c3f, 0x5133, 0x55f5,
        0x5a82, 0x5ed7, 0x62f2, 0x66cf, 0x6a6d, 0x6dca, 0x70e2, 0x73b5,
        0x7641, 0x7884, 0x7a7d, 0x7c29, 0x7d8a, 0x7e9d, 0x7f62, 0x7fd8,
        0x8000, 0x7fd8, 0x7f62, 0x7e9d, 0x7d8a, 0x7c29, 0x7a7d, 0x7884,
        0x7641, 0x73b5, 0x70e2, 0x6dca, 0x6a6d, 0x66cf, 0x62f2, 0x5ed7,
        0x5a82, 0x55f5, 0x5133, 0x4c3f, 0x471c, 0x41ce, 0x3c56, 0x36ba,
        0x30fb, 0x2b1f, 0x2528, 0x1f19, 0x18f8, 0x12c8, 0x0c8b, 0x0647,
        0x0000, 0xf9b9, 0xf375, 0xed38, 0xe708, 0xe0e7, 0xdad8, 0xd4e1,
        0xcf05, 0xc946, 0xc3aa, 0xbe32, 0xb8e4, 0xb3c1, 0xaecd, 0xaa0b,
        0xa57e, 0xa129, 0x9d0e, 0x9931, 0x9593, 0x9236, 0x8f1e, 0x8c4b,
        0x89bf, 0x877c, 0x8583, 0x83d7, 0x8276, 0x8163, 0x809e, 0x8028,
        0x8000, 0x8028, 0x809e, 0x8163, 0x8276, 0x83d7, 0x8583, 0x877c,
        0x89bf, 0x8c4b, 0x8f1e, 0x9236, 0x9593, 0x9931, 0x9d0e, 0xa129,
```

```
        0xa57e, 0xaa0b, 0xaecd, 0xb3c1, 0xb8e4, 0xbe32, 0xc3aa, 0xc946,
        0xcf05, 0xd4e1, 0xdad8, 0xe0e7, 0xe708, 0xed38, 0xf375, 0xf9b9,
    };
    static const short coeffs[8] =
    {
        0x0800, 0x1000, 0x2000, 0x4000,
        0x4000, 0x2000, 0x1000, 0x0800
    };
    int i, ok = 1;
    short y[128];
    static const short expected[128] =
    {
        0x1474, 0x1a37, 0x1fe9, 0x2588, 0x2b10, 0x307d, 0x35cc, 0x3afa,
        0x4003, 0x44e5, 0x499d, 0x4e27, 0x5281, 0x56a9, 0x5a9a, 0x5e54,
        0x61d4, 0x6517, 0x681c, 0x6ae1, 0x6d63, 0x6fa3, 0x719d, 0x7352,
        0x74bf, 0x6de5, 0x66c1, 0x5755, 0x379e, 0x379e, 0x5755, 0x66c1,
        0x6de5, 0x74bf, 0x7352, 0x719d, 0x6fa3, 0x6d63, 0x6ae1, 0x681c,
        0x6517, 0x61d4, 0x5e54, 0x5a9a, 0x56a9, 0x5281, 0x4e27, 0x499d,
        0x44e5, 0x4003, 0x3afa, 0x35cc, 0x307d, 0x2b10, 0x2588, 0x1fe9,
        0x1a37, 0x1474, 0x0ea5, 0x08cd, 0x02f0, 0xfd10, 0xf733, 0xf15b,
        0xeb8c, 0xe5c9, 0xe017, 0xda78, 0xd4f0, 0xcf83, 0xca34, 0xc506,
        0xbffd, 0xbb1b, 0xb663, 0xb1d9, 0xad7f, 0xa957, 0xa566, 0xa1ac,
        0x9e2c, 0x9ae9, 0x97e4, 0x951f, 0x929d, 0x905d, 0x8e63, 0x8cae,
        0x8b41, 0x8a1b, 0x893f, 0x88ab, 0x8862, 0x8862, 0x88ab, 0x893f,
        0x8a1b, 0x8b41, 0x8cae, 0x8e63, 0x905d, 0x929d, 0x951f, 0x97e4,
        0x9ae9, 0x9e2c, 0xa1ac, 0xa566, 0xa957, 0xad7f, 0xb1d9, 0xb663,
        0xbb1b, 0xbffd, 0xc506, 0xca34, 0xcf83, 0xd4f0, 0xda78, 0xe017,
        0xe5c9, 0xebcc, 0xf229, 0xf96a, 0x02e9, 0x0dd8, 0x1937, 0x24ce,
    };
    fir(y, x + 7, coeffs, 128, 8);
    for (i = 0; i < sizeof(y)/sizeof(*y); ++i)
    {
        if (y[i] != expected[i])
        {
            printf("mismatch: y[%d] = 0x%04x; expected[%d] = 0x%04x\n", i, y[i], i, expected[i]);
            ok = 0;
            break;
        }
    }
    if (ok) printf("** TEST PASSED OK **\n");
    return ok ? 0 : 1;
}
#ifdef __TARGET_ARCH_7_A
__asm void init_cpu() {
    // Set up CPU state
    MRC p15,0,r4,c1,c0,0
    ORR r4,r4,#0x00400000   // enable unaligned mode (U=1)
    BIC r4,r4,#0x00000002   // disable alignment faults (A=0)
    // MMU not enabled: no page tables
```

```
    MCR p15,0,r4,c1,c0,0
#ifdef __BIG_ENDIAN
    SETEND BE
#endif
    MRC p15,0,r4,c1,c0,2    // Enable VFP access in the CAR -
    ORR r4,r4,#0x00f00000   // must be done before any VFP instructions
    MCR p15,0,r4,c1,c0,2
    MOV r4,#0x40000000      // Set EN bit in FPEXC
    MSR FPEXC,r4
    IMPORT __main
    B __main
}
#endif
```

示例 3-23 DSP 向量化代码

```
/*
* DSP Vectorizable example code.
* Copyright 2006 ARM Limited. All rights reserved.
*
* Includes embedded assembly to initialize cpu; link using '--entry=init_cpu'.
*
* Build using:
*   armcc -c dsp_vector_example.c --cpu Cortex-A8 -Otime --vectorize
*   armlink -o dsp_vector_example.axf dsp_vector_example.o --entry=init_cpu
*/
#include <stdio.h>
#include "dspfns.h"
void fn(short *__restrict r, int n, const short *__restrict a, const short *__restrict b)
{
    int i;
    for (i = 0; i < n; ++i)
    {
        r[i] = add(a[i], b[i]);
    }
}
int main()
{
    static const short x[128] =
    {
        0x0000, 0x0647, 0x0c8b, 0x12c8, 0x18f8, 0x1f19, 0x2528, 0x2b1f,
        0x30fb, 0x36ba, 0x3c56, 0x41ce, 0x471c, 0x4c3f, 0x5133, 0x55f5,
        0x5a82, 0x5ed7, 0x62f2, 0x66cf, 0x6a6d, 0x6dca, 0x70e2, 0x73b5,
        0x7641, 0x7884, 0x7a7d, 0x7c29, 0x7d8a, 0x7e9d, 0x7f62, 0x7fd8,
        0x8000, 0x7fd8, 0x7f62, 0x7e9d, 0x7d8a, 0x7c29, 0x7a7d, 0x7884,
        0x7641, 0x73b5, 0x70e2, 0x6dca, 0x6a6d, 0x66cf, 0x62f2, 0x5ed7,
        0x5a82, 0x55f5, 0x5133, 0x4c3f, 0x471c, 0x41ce, 0x3c56, 0x36ba,
```

```
    0x30fb, 0x2b1f, 0x2528, 0x1f19, 0x18f8, 0x12c8, 0x0c8b, 0x0647,
    0x0000, 0xf9b9, 0xf375, 0xed38, 0xe708, 0xe0e7, 0xdad8, 0xd4e1,
    0xcf05, 0xc946, 0xc3aa, 0xbe32, 0xb8e4, 0xb3c1, 0xaecd, 0xaa0b,
    0xa57e, 0xa129, 0x9d0e, 0x9931, 0x9593, 0x9236, 0x8f1e, 0x8c4b,
    0x89bf, 0x877c, 0x8583, 0x83d7, 0x8276, 0x8163, 0x809e, 0x8028,
    0x8000, 0x8028, 0x809e, 0x8163, 0x8276, 0x83d7, 0x8583, 0x877c,
    0x89bf, 0x8c4b, 0x8f1e, 0x9236, 0x9593, 0x9931, 0x9d0e, 0xa129,
    0xa57e, 0xaa0b, 0xaecd, 0xb3c1, 0xb8e4, 0xbe32, 0xc3aa, 0xc946,
    0xcf05, 0xd4e1, 0xdad8, 0xe0e7, 0xe708, 0xed38, 0xf375, 0xf9b9,
};
static const short y[128] =
{
    0x8000, 0x7fd8, 0x7f62, 0x7e9d, 0x7d8a, 0x7c29, 0x7a7d, 0x7884,
    0x7641, 0x73b5, 0x70e2, 0x6dca, 0x6a6d, 0x66cf, 0x62f2, 0x5ed7,
    0x5a82, 0x55f5, 0x5133, 0x4c3f, 0x471c, 0x41ce, 0x3c56, 0x36ba,
    0x30fb, 0x2b1f, 0x2528, 0x1f19, 0x18f8, 0x12c8, 0x0c8b, 0x0647,
    0x0000, 0xf9b9, 0xf375, 0xed38, 0xe708, 0xe0e7, 0xdad8, 0xd4e1,
    0xcf05, 0xc946, 0xc3aa, 0xbe32, 0xb8e4, 0xb3c1, 0xaecd, 0xaa0b,
    0xa57e, 0xa129, 0x9d0e, 0x9931, 0x9593, 0x9236, 0x8f1e, 0x8c4b,
    0x89bf, 0x877c, 0x8583, 0x83d7, 0x8276, 0x8163, 0x809e, 0x8028,
    0x8000, 0x8028, 0x809e, 0x8163, 0x8276, 0x83d7, 0x8583, 0x877c,
    0x89bf, 0x8c4b, 0x8f1e, 0x9236, 0x9593, 0x9931, 0x9d0e, 0xa129,
    0xa57e, 0xaa0b, 0xaecd, 0xb3c1, 0xb8e4, 0xbe32, 0xc3aa, 0xc946,
    0xcf05, 0xd4e1, 0xdad8, 0xe0e7, 0xe708, 0xed38, 0xf375, 0xf9b9,
    0x0000, 0x0647, 0x0c8b, 0x12c8, 0x18f8, 0x1f19, 0x2528, 0x2b1f,
    0x30fb, 0x36ba, 0x3c56, 0x41ce, 0x471c, 0x4c3f, 0x5133, 0x55f5,
    0x5a82, 0x5ed7, 0x62f2, 0x66cf, 0x6a6d, 0x6dca, 0x70e2, 0x73b5,
    0x7641, 0x7884, 0x7a7d, 0x7c29, 0x7d8a, 0x7e9d, 0x7f62, 0x7fd8,
};
short r[128];
static const short expected[128] =
{
    0x8000, 0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff,
    0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff,
    0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff,
    0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff, 0x7fff,
    0x8000, 0x7991, 0x72d7, 0x6bd5, 0x6492, 0x5d10, 0x5555, 0x4d65,
    0x4546, 0x3cfb, 0x348c, 0x2bfc, 0x2351, 0x1a90, 0x11bf, 0x08e2,
    0x0000, 0xf71e, 0xee41, 0xe570, 0xdcaf, 0xd404, 0xcb74, 0xc305,
    0xbaba, 0xb29b, 0xaaab, 0xa2f0, 0x9b6e, 0x942b, 0x8d29, 0x866f,
    0x8000, 0x8000, 0x8000, 0x8000, 0x8000, 0x8000, 0x8000, 0x8000,
    0x8000, 0x8000, 0x8000, 0x8000, 0x8000, 0x8000, 0x8000, 0x8000,
    0x8000, 0x8000, 0x8000, 0x8000, 0x8000, 0x8000, 0x8000, 0x8000,
    0x8000, 0x8000, 0x8000, 0x8000, 0x8000, 0x8000, 0x8000, 0x8000,
    0x8000, 0x866f, 0x8d29, 0x942b, 0x9b6e, 0xa2f0, 0xaaab, 0xb29b,
    0xbaba, 0xc305, 0xcb74, 0xd404, 0xdcaf, 0xe570, 0xee41, 0xf71e,
    0x0000, 0x08e2, 0x11bf, 0x1a90, 0x2351, 0x2bfc, 0x348c, 0x3cfb,
    0x4546, 0x4d65, 0x5555, 0x5d10, 0x6492, 0x6bd5, 0x72d7, 0x7991,
```

```
    };
    int i, ok = 1;

    fn(r, sizeof(r)/sizeof(*r), x, y);

    for (i = 0; i < sizeof(r)/sizeof(*r); ++i)
    {
        if (r[i] != expected[i])
        {
            printf("mismatch: r[%d] = 0x%04x; expected[%d] = 0x%04x\n", i, r[i], i, expected[i]);
            ok = 0;
            break;
        }
    }
    if (ok) printf("** TEST PASSED OK **\n");
        return ok ? 0 : 1;
}
#ifdef __TARGET_ARCH_7_A
__asm void init_cpu()
{
    // Set up CPU state
    MRC p15,0,r4,c1,c0,0
    ORR r4,r4,#0x00400000   // enable unaligned mode (U=1)
    BIC r4,r4,#0x00000002   // disable alignment faults (A=0)
    // MMU not enabled: no page tables
    MCR p15,0,r4,c1,c0,0
#ifdef __BIG_ENDIAN
    SETEND BE
#endif
    MRC p15,0,r4,c1,c0,2    // Enable VFP access in the CAR -
    ORR r4,r4,#0x00f00000   // must be done before any VFP instructions
    MCR p15,0,r4,c1,c0,2
    MOV r4,#0x40000000      // Set EN bit in FPEXC
    MSR FPEXC,r4
    IMPORT __main
    B __main
}
#endif
```

第 4 章
编译器功能

本章概述了编译器的 ARM 特有的功能。本章分为以下几节：

4.1 内在函数

编译器支持多个内在函数系列，其中包括：

- 用于在 C 和 C++ 代码中实现 ARM、Thumb 和 NEON 指令的指令内在函数
- 用于实现 ETSI 基本运算的内在函数
- 用于仿真 TI C55x 编译器内在函数的内在函数
- 与 NEON 向量化编译器一起使用的 NEON™ 内在函数。

本节将介绍这些内在函数系列。

4.1.1 关于内在函数

C 和 C++ 虽然适用于各种各样的任务，但不提供对特定应用领域的内置支持，例如，*数字信号处理* (DSP)。

在给定的应用领域，通常需要经常执行一系列特定于应用领域的运算。尽管如此，通常还是不能在 C 或 C++ 中有效进行这些运算。一个典型示例是对两个 32 位有符号二进制补码整数进行饱和相加，这常用于 DSP 编程。示例 4-1 是以 C 实现的该典型示例。

示例 4-1 饱和加法运算的 C 实现

```
#include <limits.h>
int L_add(const int a, const int b)
{
    int c;
    c = a + b;
    if (((a ^ b) & INT_MIN) == 0)
    {
        if ((c ^ a) & INT_MIN)
        {
            c = (a < 0) ? INT_MIN : INT_MAX;
        }
    }
    return c;
}
```

*内在函数*提供了一种简便的手段，可以在 C 和 C++ 源代码中合并应用领域特定的运算，而不必借助复杂的实现，例如，在嵌入式汇编器或内联汇编器中。内在函数的形式与 C 或 C++ 中的函数调用一样，不过，在编译过程中，它将被特定的低级指令序列所代替。例如，在使用内在函数实现时，第4-2 页的示例 4-1 的饱和加法函数的形式为：

```
#include <dspfns.h>    /* Include ETSI intrinsics */
...
int a, b, result;
...
result = L_add(a, b);  /* Saturated add of a and b */
```

使用内在函数可提供以下性能优势：

- 代替内在函数的底层指令比对应的 C 或 C++ 实现更有效，可减少指令和周期计数。为了执行内在函数，编译器将自动为指定目标体系结构生成最佳的指令序列。例如，`L_add` 内在函数直接映射到 ARM v5TE 汇编语言指令 `qadd`：

  ```
  QADD r0, r0, r1    /* Assuming r0 = a, r1 = b on entry */
  ```

- 向编译器提供的信息比基础 C 和 C++ 语言所能传达的更多。这样，编译器可以执行优化，生成以这种方式才能实现的指令序列。

对于实时处理应用程序，这些性能优势是至关重要的。不过也需要谨慎，因为使用内在函数可能会降低代码的可移植性。

4.1.2 指令内在函数

ARM 编译器提供了一系列指令内在函数，可用于在 C 或 C++ 代码中实现 ARM 汇编语言指令。总之，通过这些内在函数，可以结合使用 C 代码和指令内在函数来仿真内联汇编代码。

通用内在函数

《编译器参考指南》介绍了属于 ISO C 和 C++ 标准的 ARM 语言扩展的以下通用内在函数：

- 第4-71 页的*__breakpoint*
- 第4-74 页的*__current_pc*
- 第4-74 页的*__current_sp*
- 第4-83 页的*__nop*
- 第4-89 页的*__return_address*
- 第4-91 页的*__semihost*

另请参阅 *《编译器参考指南》*中第4-112 页的*GNU 内置函数*。

所有体系结构都提供这些内在函数的实现。

控制 IRQ 和 FIQ 中断的内在函数

《编译器参考指南》介绍了可以用于控制 IRQ 和 FIQ 中断的以下内在函数：

- 第4-76 页的*__disable_irq*
- 第4-77 页的*__enable_irq*
- 第4-75 页的*__disable_fiq*
- 第4-77 页的*__enable_fiq*

不能使用这些内在函数更改其他任何 CPSR 位，包括模式、状态和不精确数据中止设置。这意味着只有当处理器已处于特权模式时才能使用内在函数，因为在用户模式下无法更改 CPSR 和 SPSR 的控制位。

在 ARM 和 Thumb 状态下，这些内在函数可用于所有处理器体系结构：

- 如果要针对支持 ARMv6 （或更高版本）的处理器进行编译，则将为这些函数内联生成 CPS 指令，例如：

  ```
  CPSID  i
  ```

- 如果要针对在 ARM 状态下支持 ARMv4 或 ARMv5 的处理器进行编译，则编译器将内联 MRS 和 MSR 指令序列，例如：

  ```
  MRS  r0, CPSR
  ORR  r0, r0, #0x80
  MSR  CPSR_c, r0
  ```

- 如果要针对在 Thumb 状态下支持 ARMv4 或 ARMv5 的处理器进行编译，则编译器将调用辅助函数，例如：

  ```
  BL    __ARM_disable_irq
  ```

有关这些指令的详细信息，请参阅 《汇编器指南》。

用于插入优化屏障的内在函数

ARM 编译器可以执行一系列优化，包括重新排序指令以及合并某些运算。在某些情况下，例如多个进程同时访问内存的系统级编程，有必要禁用指令重新排序并强制更新内存。

以下优化屏障内在函数不生成代码，但会导致代码大小略有增加，并会导致更多内存访问。请参阅 《编译器参考指南》中的以下内容：

- 第4-91 页的*__schedule_barrier*
- 第4-79 页的*__force_stores*
- 第4-83 页的*__memory_changed*

注意

在某些系统中，内存屏障内在函数可能不足以保证内存一致性。例如，`__memory_changed()` 内在函数强制将寄存器中保留的值写出到内存。不过，如果数据的目标保存在可缓冲的区域中，则它可以在写缓冲区中等待。在这种情况下，可能还需要写入 CP15 或使用内存屏障指令清空写缓冲区。有关详细信息，请参阅 ARM 处理器的技术参考手册。

用于插入本机指令的内在函数

使用以下内在函数可以将 ARM 处理器指令插入编译器生成的指令流。请参阅《编译器参考指南》中的以下内容：

- 第4-72 页的*__cdp*
- 第4-73 页的*__clrex*
- 第4-79 页的*__ldrex*
- 第4-81 页的*__ldrt*
- 第4-84 页的*__pld*
- 第4-86 页的*__pli*
- 第4-88 页的*__rbit*
- 第4-89 页的*__rev*
- 第4-90 页的*__ror*
- 第4-93 页的*__sev*
- 第4-95 页的*__strex*
- 第4-98 页的*__strt*
- 第4-99 页的*__swp*
- 第4-101 页的*__wfe*

- 第4-102 页的__wfi
- 第4-102 页的__yield

用于数字信号处理的内在函数

《编译器参考指南》中介绍的以下内在函数可以协助实现 DSP 算法：

- 第4-73 页的__clz
- 第4-78 页的__fabs
- 第4-78 页的__fabsf
- 第4-87 页的__qadd
- 第4-87 页的__qdbl
- 第4-88 页的__qsub
- 第4-93 页的__sqrt
- 第4-94 页的__sqrtf
- 第4-94 页的__ssat
- 第4-100 页的__usat

另请参阅 *《编译器参考指南》*中第4-103 页的*ARMv6 SIMD 内在函数*。

这些内在函数为以下体系结构引入相应目标指令：

- ARM v5TE 及更高版本的 ARM 体系结构
- Thumb-2 体系结构 （“M”版本除外）。

不是每个指令都有自己的内在函数。编译器可以组合多个内在函数或内在函数与 C 运算符的组合来生成更强大的指令。例如，ARM5TE QDADD 指令是由 __qadd 和 __qdbl 组合实现的。

4.1.3 ETSI 基本运算

欧洲电信标准协会 (ETSI) 制定了多个语音编码建议书，例如，G.723.1 和 G.729 建议书。这些建议书包括多媒体数字信号编解码器的参考执行的源代码和测试序列。

ETSI 提供的语音编解码器的模型执行以一组称为 *ETSI 基本运算*的 C 函数为基础。ETSI 基本运算包括饱和算法的 16 位、32 位和 40 位运算、16 位和 32 位逻辑运算以及数据类型转换的 16 位和 32 位运算。

注意

2.0 版的 ETSI 基本运算集合 （如 《ITU-T 软件工具库 2005 用户手册》 *(ITU-T Software Tool Library 2005 User's manual)* 所述）引入了 16 位、 32 位和 40 位运算。RVCT 不支持这些运算。

ETSI 基本运算是一组供发布编解码器算法的开发人员使用的基元，而不是供开发人员在 C 或 C++ 中实现编解码器所用的库。RVCT 通过头文件 `dspfns.h` 提供对 ETSI 基本运算的支持。

RVCT 中的 ETSI 运算

`dspfns.h` 头文件包含 ETSI 基本运算的定义，其形式为 C 代码和内在函数的组合。RVCT 支持原始 ETSI 基本运算系列 （如 ETSI G.729 建议书 *《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》* 所述），其中包括：

- 16 位和 32 位饱和算术运算，例如 `add` 和 `sub`。例如， `add(v1, v2)` 使用溢出控制和饱和算法，将两个 16 位数字 `v1` 和 `v2` 相加，返回 16 位结果。
- 16 位和 32 位乘法运算（如 `mult` 和 `L_mult`）。例如，`mult(v1, v2)` 将两个 16 位数字 `v1` 和 `v2` 相乘，返回换算的 16 位结果。
- 16 位算术移位运算 （如 `shl` 和 `shr`）。例如，饱和左移运算 `shl(v1, v2)` 在算术上将 16 位输入 `v1` 向左移 `v2` 个位置。如果移位计数为负，则会将 `v1` 向右移 `v2` 个位置。
- 16 位数据转换运算 （如 `extract_l`、 `extract_h` 和 `round`）。例如， `round(L_v1)` 使用饱和算法舍去 32 位输入 `L_v1` 的低 16 位，舍入到最高有效 16 位。

  注意

  请注意， `dspfns.h` 头文件和 ISO C99 头文件 `math.h` 都定义了 （不同版本的）函数 `round()`。请留意避免出现此潜在冲突。

有关 RVCT 支持的 ETSI 基本运算的完整列表，请参阅头文件 `dspfns.h`。

此外，请参阅：

- 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)*)。

要获取这些文档，请访问 ITU-T （ITU 通信局）网站 http://www.itu.int。

溢出和进位

dspfns.h 中的 ETSI 基本运算执行会公开状态标记 Overflow 和 Carry。这些标记可以作为全局变量在 C 或 C++ 程序使用。例如：

```
#include <dspfns.h>                 /* include ETSI intrinsics */
#include <stdio.h>
...
const int BUFLEN=255;
int a[BUFLEN], b[BUFLEN], c[BUFLEN];
...
Overflow = 0;                       /* clear overflow flag */
for (i = 0; i < BUFLEN; ++i) {
    c[i] = L_add(a[i], b[i]);       /* saturated add of a[i] and b[i] */
}
if (Overflow)
{
    fprintf(stderr, "Overflow on saturated addition\n");
}
```

通常，饱和函数对溢出具有粘着效果。也就是说，在未显式清除前，溢出标记将保持设置状态。有关详细信息，请参阅头文件 dspfns.h。

4.1.4 TI C55x 内在函数

德州仪器 (TI) C55x 编译器可以识别许多用于优化 C 代码的内在函数。RVCT 支持通过头文件 c55x.h 对选定 TI C55x 内在函数进行仿真。在 c55x.h 中仿真的 TI C55x 内在函数包括：

- 加法、减法、求反和求绝对值的内在函数，例如 _sadd 和 _ssub。例如，_sadd(v1, v2) 将返回 v1 和 v2 的 16 位饱和求和值。
- 乘法和移位的内在函数，例如， _smpy 和 _sshl。例如， _smpy(v1, v2) 将返回 v1 和 v2 的饱和小数模式乘积值。

- 舍入、饱和、位计数和求极值的内在函数，例如 _round 和 _count。例如，_round(v1) 使用饱和算法将值 v1 与 215 相加然后清除低 16 位，返回舍入的值。

c55x.h 中不支持以下 TI C55x 内在函数：

- 加法和乘加的内在函数的相关变体。这包括所有以 _a_ 为前缀的 TI C55x 内在函数，例如，_a_sadd 和 _a_smac。
- 乘法和移位的内在函数的舍入变体，例如，_smacr 和 _smasr。
- 内在函数的所有 **long long** 变体。这包括所有以 _ll 为前缀的 TI C55x 内在函数，例如，_llsadd 和 _llshl。由于内在函数的 **long long** 变体对 40 位数据执行运算，因此 RVCT 不支持此类变体。
- 所有具有副作用的算术内在函数。例如，c55x.h 中未定义 TI C55x 内在函数 _firs 和 _lms。
- ETSI 支持函数的内在函数，例如，L_add_c 和 L_sub_c。

  注意

  饱和除法 divs 的 ETSI 支持函数例外。c55x.h 中支持此内在函数。

有关 RVCT 中仿真的 TI C55x 内在函数的完整列表，请参阅头文件 c55x.h。

有关 TI 编译器内在函数的详细信息，请访问 http://www.ti.com。

4.1.5 已命名的寄存器变量

通过编译器可以使用已命名的寄存器变量来访问基于 ARM 体系结构的处理器的寄存器。

已命名的寄存器变量是通过结合使用 **register** 关键字与 __asm 关键字声明的。__asm 关键字使用一个参数 （一个字符串），该参数对寄存器进行命名。例如，声明：

```
register int R0 __asm("r0");
```

将 R0 声明为寄存器 r0 的命名寄存器变量。若要详细了解可使用命名寄存器变量访问的、基于 ARM 体系结构的处理器的寄存器，请参阅 *《编译器参考指南》*中第4-108 页的*已命名的寄存器变量*。

命名寄存器变量的典型用法是访问*应用程序状态寄存器* (APSR) 中的位 （请参阅 *《汇编器指南》*中第2-7 页的*应用程序状态寄存器 (APSR)*）。示例 3-3 演示使用命名寄存器变量在 APSR 中设置饱和标记 Q。

示例 4-2 使用命名寄存器变量在 APSR 中设置位

```
#ifndef __BIG_ENDIAN // bitfield layout of APSR is sensitive to endianness
typedef union
{
    struct
    {
        int mode:5;
        int T:1;
        int F:1;
        int I:1;
        int _dnm:19;
        int Q:1;
        int V:1;
        int C:1;
        int Z:1;
        int N:1;
    } b;
    unsigned int word;
} PSR;
#else /* __BIG_ENDIAN */
typedef union
{
    struct
    {
        int N:1;
        int Z:1;
        int C:1;
        int V:1;
        int Q:1;
        int _dnm:19;
        int I:1;
        int F:1;
        int T:1;
        int mode:5;
    } b;
    unsigned int word;
} PSR;
#endif /* __BIG_ENDIAN */
register PSR apsr __asm("apsr");
void set_Q(void)
```

```
{
    apsr.b.Q = 1;
}
```

4.1.6 NEON 内在函数

ARM 编译器提供了 NEON 内在函数，可在向量化编译器和编写汇编程序代码之间提供一个生成 SIMD 代码的中间步骤。与直接编写汇编程序代码相比，使用此功能可以更轻松地编写利用 NEON 体系结构的代码。

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。有关 NEON 内在函数的详细信息，请参阅 *《编译器参考指南》*中的附录 E *使用 NEON 支持*。

示例 4-3 是一个使用 NEON 内在函数的简短示例。生成示例：

1. 使用以下选项编译 C 文件 neon_example.c：
   ```
   armcc -c --debug --cpu=Cortex-A8 neon_example.c
   ```
2. 使用以下命令链接映像：
   ```
   armlink neon_example.o -o neon_example.axf
   ```
3. 使用兼容的调试器 （例如 RealView Debugger）加载并运行映像。

示例 4-3 NEON 内在函数

```
/* neon_example.c - Neon intrinsics example program */
#include <stdint.h>
#include <stdio.h>
#include <assert.h>
#include <arm_neon.h>
/* fill array with increasing integers beginning with 0 */
void fill_array(int16_t *array, int size)
{    int i;
    for (i = 0; i < size; i++)
    {
         array[i] = i;
    }
}
/* return the sum of all elements in an array. This works by calculating 4
totals (one for each lane) and adding those at the end to get the final total */
int sum_array(int16_t *array, int size)
{
     /* initialize the accumulator vector to zero */
```

```
    int16x4_t acc = vdup_n_s16(0);
    int32x2_t acc1;
    int64x1_t acc2;
    /* this implementation assumes the size of the array is a multiple of 4 */
    assert((size % 4) == 0);
    /* counting backwards gives better code */
    for (; size != 0; size -= 4)
    {
        int16x4_t vec;
        /* load 4 values in parallel from the array */
        vec = vld1_s16(array);
        /* increment the array pointer to the next element */
        array += 4;
        /* add the vector to the accumulator vector */
        acc = vadd_s16(acc, vec);
     }
     /* calculate the total */
     acc1 = vpaddl_s16(acc);
     acc2 = vpaddl_s32(acc1);
     /* return the total as an integer */
     return (int)vget_lane_s64(acc2, 0);
}
/* main function */
int main()
{
     int16_t my_array[100];
     fill_array(my_array, 100);
     printf("Sum was %d\n", sum_array(my_array, 100));
     return 0;
}
```

有关 NEON 的详细信息，请参阅：

- 第2-23 页的*NEON 技术*
- *《汇编器指南》*。

4.2 编译指示

ARM 编译器可以识别以下格式的编译指示：

```
#pragma no_feature-name

#pragma feature_name
```

——注意——

编译指示优先于相关的命令行选项。例如，`#pragma arm` 优先于 `--thumb` 命令行选项。

有关详细信息，请参阅 *《编译器参考指南》* 中的相关章节：

用于保存和恢复编译指示状态的编译指示

以下编译指示用于保存和恢复编译指示状态：

- 第4-67 页的*#pragma pop*
- 第4-67 页的*#pragma push*

用于控制优化目标的编译指示

这些编译指示用于向各个函数分配优化目标。编译指示必须放置在函数外部。以下编译指示控制这些优化：

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

控制代码生成的编译指示

以下编译指示控制代码的生成方式：

- 第4-56 页的*#pragma arm*
- 第4-70 页的*#pragma thumb*
- 第4-61 页的*#pragma exceptions_unwind*, *#pragma no_exceptions_unwind*

控制循环展开的编译指示：

以下编译指示控制循环的展开方式：

- 第4-68 页的*#pragma unroll [(n)]*
- 第4-69 页的*#pragma unroll_completely*

控制*预编译头文件* (PCH) 处理的编译指示

以下编译指示控制 PCH 处理：

- 第4-62 页的*#pragma hdrstop*
- 第4-63 页的*#pragma no_pch*

控制匿名结构和联合的编译指示

以下编译指示控制匿名结构和联合的使用：

- 第4-56 页的*#pragma anon_unions*, *#pragma no_anon_unions*

控制诊断消息的编译指示

以下编译指示控制在消息编号中具有 -D 后缀的诊断消息输出：

- 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,...]*
- 第4-61 页的*#pragma diag_warning tag[, tag, ...]*

其他编译指示

- 第4-56 页的*#pragma arm section [section_sort_list]*
- 第4-62 页的*#pragma import(__use_full_stdio)*
- 第4-63 页的*#pragma inline*, *#pragma no_inline*
- 第4-64 页的*#pragma once*
- 第4-65 页的*#pragma pack(n)*
- 第4-67 页的*#pragma softfp_linkage*, *#pragma no_softfp_linkage*
- 第4-62 页的*#pragma import symbol_name*

4.3 位处理操作

本节介绍编译器如何支持位处理操作功能。

注意

位处理操作是 Cortex-M3 处理器的一种功能和一些派生。此功能在其他 ARM 处理器上不可用。

有关位处理操作的体系结构支持的详细信息，请参阅处理器的 《技术参考手册》 (*Technical Reference Manual*)。

处理器通过以下方式来支持位处理操作：

- `__attribute__((bitband))` 语言扩展
- `--bitband` 命令行选项。

4.3.1 使用 `__attribute__((bitband))`

`__attribute__((bitband))` 是用于对结构的类型定义进行位处理操作的类型属性。请参阅示例 4-4 和第4-16 页的示例 4-5。

示例 4-4 未定位的对象

```
/* foo.c */

typedef struct {
  int i : 1;
  int j : 2;
  int k : 3;
} BB __attribute__((bitband));

BB value; // Unplaced object

void update_value(void)
{
    value.i = 1;
    value.j = 0;
}

/* end of foo.c */
```

在第4-15 页的示例 4-4 中，必须将未定位的位处理操作对象重新定位到位处理操作区。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》*。

或者，您可使用 __attribute__((at())) 将位处理操作对象放置在位处理操作区的特定地址处。请参阅示例 4-5。

示例 4-5 已定位的对象

```
/* foo.c */

typedef struct {
  int i : 1;
  int j : 2;
  int k : 3;
} BB __attribute__((bitband));

BB value __attribute__((at(0x20000040))); // Placed object

void update_value(void)
{
    value.i = 1;
    value.j = 0;
}

/* end of foo.c */
```

有关详细信息，请参阅 *《编译器参考指南》*中第4-42 页的 *__attribute__((bitband))* 和第4-48 页的 *__attribute__((at(address)))*。

4.3.2 使用命令行上的 --bitband

--bitband 命令行选项将对所有非 **const** 全局结构对象执行位处理操作。

在第4-17 页的示例 4-6 中，对 foo.c 应用--bitband 时，将在写入 value.i 时执行位处理操作，即，值 0x00000001 将被写入 value.i 映射到位处理操作区中的位处理操作别名字。

不在访问 value.j 和 value.k 时执行位处理操作。

示例 4-6 使用 --bitband 命令行选项

```
/* foo.c */

typedef struct {
  int i : 1;
  int j : 2;
  int k : 3;
} BB;

BB value __attribute__((at(0x20000040))); // Placed object

void update_value(void)
{
    value.i = 1;
    value.j = 0;
}

/* end of foo.c */
```

armcc 支持对通过绝对地址访问的对象执行位处理操作。在示例 4-7 中，对 foo.c 应用 --bitband 时，将在访问 rts 时执行位处理操作。

示例 4-7 对通过绝对地址访问的对象执行位处理操作

```
/* foo.c */

typedef struct {
  int rts : 1;
  int cts : 1;
  unsigned int data;
} uart;

#define com2 (*((volatile uart *)0x20002000))
void put_com2(int n)
{
  com2.rts = 1;
  com2.data = n;
}

/* end of foo.c */
```

有关详细信息，请参阅 *《编译器参考指南》*中第2-17 页的*--bitband*。

4.3.3 限制

应用的限制如下：

- 位处理操作只能与 **struct** 类型一起使用。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。

4.4 线程局部存储

线程局部存储 (TLS) 是静态存储类，与堆栈类似，对于每个执行线程都是私有的。进程中的每个线程都有一个可以存储线程特定数据的位置。系统将分配变量，使现有的每个线程都有一个变量实例。

每个线程终止之前，都会释放其动态内存，指向该线程中的线程局部变量的所有指针都将失效。

请参阅 *《编译器参考指南》*中第4-29 页的*__declspec(thread)*。

4.5 8 字节对齐功能

ARM 编译器具有下列 8 字节对齐功能：

- 《ARM 体系结构的过程调用标准》 (*Procedure Call Standard for the ARM Architecture*) (AAPCS) 要求堆栈在所有外部接口上都是 8 字节对齐的。ARM 编译器和 C 库保留堆栈的 8 字节对齐。此外，缺省的 C 库内存模型保持堆的 8 字节对齐。
- 代码的编译方式要求并保持了在外部接口的 8 字节对齐约束。
- 如果项目包含汇编文件、旧对象文件或库，则需要检查它们是否保持 8 字节堆栈对齐，并根据需要予以纠正。请参阅 《汇编器指南》和 《链接器用户指南》。
- 在 RVCT v2.0 和更高版本中，**`double`** 和 **`long long`** 数据类型为 8 字节对齐。这样，可高效地在 ARMv5TE 和更高版本中使用 `LDRD` 和 `STRD` 指令。
- `malloc()`、`realloc()` 和 `calloc()` 的缺省实现维护一个 8 字节对齐的堆。
- `alloca()` 的缺省实现返回一个 8 字节对齐的内存块。有关该 C 库扩展的详细信息，请参阅 *《库指南》*中第2-66 页的*alloca()*。

第 5 章
编程惯例

ARM 编译器 armcc 是一种成熟的、工业级 ISO C 和 C++ 编译器，可生成高度优化的高质量机器代码。通过使用在 RISC 处理器（如 ARM 内核）上能良好运行的编程惯例和技巧，可以提高您的 C 和 C++ 源代码的可移植性、效率和耐用性。本章介绍了一些这样的编程惯例以及一些 ARM 处理器特有的编程技巧。

本章分为以下几节：

5.1 优化代码

ARM 编译器高度优化，具有代码小、性能高的特点。该编译器也可执行其他优化编译器上常用的优化功能，例如，数据流优化，包括公共子表达式排除和循环优化 （如循环组合和分发）。此外，该编译器还可执行一系列基于 ARM 体系结构的处理器特有的优化功能。

即使编译器已高度优化，通常仍可通过选择正确的优化标准、目标处理器和体系结构、内联选项来显著提高 C 或 C++ 代码的性能。

5.1.1 大小与速度的优化

编译器提供了两个选项用于优化代码大小和性能：

`-Ospace` 此选项使编译器主要针对代码大小进行优化。这是缺省选项。

`-Otime` 此选项使编译器主要针对速度进行优化。

要获得最佳效果，必须使用最合适的命令行选项来生成应用程序。

注意

这些命令行选项指示编译器使用优化，在大多数情况下能实现预期的优化效果。但是，不能保证 `-Otime` 总能生成更快的代码，也不能保证 `-Ospace` 总能生成更小的代码。

请参阅：

- *《编译器参考指南》*中第2-96 页的*-Ospace*
- *《编译器参考指南》*中第2-97 页的*-Otime*。

5.1.2 优化级别和调试视图

由编译器执行的精确优化既取决于所选的优化级别，也取决于是优化性能还是优化代码大小。

编译器支持以下优化级别：

`-O0` 最低优化。编译器执行不会影响调试视图的简单优化。

启用调试时，此选项给出可能的最佳调试视图。

`-O1` 受限优化。

启用调试时，此选项给出具有良好代码密度并且通常令人满意的调试视图。

`-O2` 高度优化。这是缺省设置。

启用调试时，此选项可能给出不太令人满意的调试视图。

-O3　　最大优化。这是可用的最积极的优化格式。如果指定此选项，则当在命令行中指定多个文件时，缺省启用多文件编译。

启用调试时，此选项通常给出较差的调试视图。

因为优化会影响对象代码到源代码的映射，所以对优化级别和 -Ospace/-Otime 的选择通常会影响调试视图。使用 --debug 启用调试时，请使用 -Onum 命令行选项显式指定最适合的优化级别。

需要简单调试视图时，选项 -O0 是可以使用的最佳选项。选择 -O0 通常会将 ELF 映像的大小增加 7% - 15%。若要减小调试表的大小，请使用 --no_debug_macros 选项。

请参阅：

- *《编译器参考指南》*中第2-36 页的*--debug, --no_debug*
- *《编译器参考指南》*中第2-37 页的*--debug_macros, --no_debug_macros*
- *《编译器参考指南》*中第2-51 页的*--dwarf2*
- *《编译器参考指南》*中第2-51 页的*--dwarf3*
- *《编译器参考指南》*中第2-94 页的*-Onum*。

5.1.3 选择目标 CPU

通常， ARM 体系结构的每个新版本都支持附加的指令、附加的运算模式、管道差异以及寄存器重命名。

- 如果编译的程序要运行在基于特定 ARM 体系结构的处理器上，最好使用 --cpu 命令行选项选择目标处理器。这使编译器可以充分利用处理器所支持的指令，并且执行特定于处理器的优化，如指令调度。
- 当编译的程序要运行在其他 ARM 处理器上时，必须通过 --cpu 命令行选项选择适合应用程序的最小公分母体系结构。例如，要为支持 ARM v6 体系结构的处理器编译代码，请使用命令行选项 --cpu 6。

注意

使用命令行选项 --cpu list 可以列出编译器支持的所有处理器和体系结构。

请参阅：

- 第2-23 页的*指定目标处理器或体系结构*
- *《编译器参考指南》*中第2-30 页的*--cpu=list*

- 《编译器参考指南》中第2-30 页的*--cpu=name*。

5.1.4 优化循环

循环是大多数程序中常用的结构。因为循环通常要消耗大量的执行时间，因此有必要对时间要求严格的循环加以注意。

循环终止

如果在编写时未加注意，循环终止条件会造成显著的开销。请尽可能：

- 始终编写计数递减到零的循环并使用简单终止条件
- 始终使用 **unsigned int** 类型的计数器，并且测试不等于零。

表 5-1 显示了一个计算 n! 例程的两个示例执行，用于说明循环终止的开销。第一个执行使用递增循环计算 n!，第二个例程使用递减循环计算 n!。

表 5-1 递增和递减循环的 C 代码

递增循环	递减循环

`int fact1(int n)`
`{`
`  int i, fact = 1;`
`  for (i = 1; i <= n; i++)`
`  fact *= i;`
`  return (fact);`
`}`
	`int fact2(int n)`
	`{`
	`  unsigned int i, fact = 1;`
	`  for (i = n; i != 0; i--)`
	`  fact *= i;`
	`  return (fact);`
	`}`

表 5-2 是编译器为第5-4 页的表 5-1 中的每个示例实现生成的机器代码的反汇编，其中，两个示例实现的 C 代码均使用选项 -O2 -Otime 进行了编译。

表 5-2 递增和递减循环的 C 反汇编

递增循环	递减循环

fact1 PROC
    MOV   r2, r0
    MOV   r0, #1
    CMP   r2, #1
    MOV   r1, r0
    BXLT   lr
|L1.20|
    MUL   r0, r1, r0
    ADD   r1, r1, #1
    CMP   r1, r2
    BLE |L1.20|
    BX   lr
    ENDP
	fact2 PROC
	    MOVS r1, r0
	    MOV r0, #1
	    BXEQ lr
	|L1.12|
	    MUL r0, r1, r0
	    SUBS r1, r1, #1
	    BNE |L1.12|
	    BX lr
	ENDP

比较表 5-2 的反汇编，可以看到，递增循环反汇编中有 ADD/CMP 指令对，而在递减循环反汇编中，则是一条 SUBS 指令。这是因为可以通过优化来去除与零的比较。

除了能节省循环中的指令外，变量 n 在循环中不需要保存，因此在递减循环反汇编中还节省了寄存器的使用。这有利于寄存器分配。

这种将循环计数器初始化为迭代所需的数字然后递减到零的技巧，也可以应用于 **while** 和 **do** 语句。

循环展开

展开小的循环虽然可以获得较高的性能，但缺点是会增加代码大小。展开循环后，对循环计数器的更新次数减少，从而执行跳转的次数减少。如果循环只迭代数次，则可以完全展开，从而完全消除循环的开销。ARM 编译器在使用 -O3 -Otime 时自动展开循环。否则，任何展开均必须在源代码中执行。

——注意——

手动展开循环可能会妨碍循环的自动重新展开和编译器的其他循环优化。

使用第5-5 页的表 5-3 中演示的两个示例例程可以说明循环展开的优缺点。两个例程均通过提取最低位并计数来高效测试单个位，在此之后将位移出。

第一个执行使用循环来对位计数。第二个例程是第一个展开四次并通过将 n 的四次移位组合为一次优化后的结果。展开通常会提供新的优化机会。

表 5-3 未展开和展开的位计数循环的 C 代码

位计数循环

```
int countbit1(unsigned int n)
{
    int bits = 0;
    while (n != 0)
    {
        if (n & 1) bits++;
        n >>= 1;
    }
    return bits;
}
```

展开的位计数循环

```
int countbit2(unsigned int n)
{
    int bits = 0;
    while (n != 0)
    {
        if (n & 1) bits++;
        if (n & 2) bits++;
        if (n & 4) bits++;
        if (n & 8) bits++;
        n >>= 4;
    }
    return bits;
}
```

表 5-4 是编译器为表 5-3 中的每个示例实现生成的机器代码的反汇编，其中，每个实现的 C 代码均使用选项 -O2 进行了编译。

表 5-4 未展开和展开的位计数循环的反汇编

位计数循环

```
countbit1 PROC
    MOV      r1, #0
    B        |L1.20|
|L1.8|
    TST      r0, #1
    ADDNE    r1, r1, #1
    LSR      r0, r0, #1
|L1.20|
    CMP      r0, #0
    BNE      |L1.8|
    MOV      r0, r1
    BX       lr
ENDP
```

展开的位计数循环

```
countbit2 PROC
    MOV      r1, r0
    MOV      r0, #0
    B        |L1.48|
|L1.12|
    TST      r1, #1
    ADDNE    r0, r0, #1
    TST      r1, #2
    ADDNE    r0, r0, #1
    TST      r1, #4
    ADDNE    r0, r0, #1
    TST      r1, #8
    ADDNE    r0, r0, #1
    LSR      r1, r1, #4
|L1.48|
    CMP      r1, #0
    BNE      |L1.12|
    BX       lr
ENDP
```

在 ARM7 上，在左列所示的位计数循环的反汇编中，检查单个位要占用六个周期。代码的大小仅为九个指令。位计数循环的展开版本一次检查四位，平均每一位只占用三个周期。不过，开销是代码大小较大，有十五条指令。

5.1.5 使用 volatile

有时，在较高的优化级别 -O2 和 -O3 上编译代码会遇到问题。例如，在轮询硬件时循环可能会卡住，或者多线程的代码可能表现出奇怪的行为。在这种情况下，您可能需要将部分变量声明为 **volatile**。

将变量声明为 **volatile** 会通知编译器该变量可以随时在执行之外进行修改，例如，由操作系统或硬件修改。因为 **volatile** 限定的变量值可以随时更改，所以只要在代码中引用变量，就一定会访问内存中变量的物理地址。这意味着编译器不能对变量执行优化，例如，将其高速缓存到本地寄存器中以避免内存访问。

与之相对的是，不将变量声明为 **volatile** 时，编译器可以假定其值不能从执行之外修改。因此，编译器可以对变量进行优化。

volatile 关键字的用法在表 5-5 中的两个示例例程中加以说明，两者均循环读取缓冲区直至将状态标记 buffer_full 设置为“true”。两个例程均假定 buffer_full 的状态可以使用程序流异步更改。

第一个例程显示了简单的循环执行。请注意，未在本执行中将变量 buffer_full 限定为 **volatile**。与之相对的是，第二个例程显示了相同的循环，但在执行中将 buffer_full 正确限定为 **volatile**。

表 5-5 非易失性和易失性缓冲区循环的 C 代码

非易失性版本的缓冲区循环	易失性版本的缓冲区循环

`int buffer_full;`
`int read_stream(void)`
`{`
`  int count = 0;`
`  while (!buffer_full)`
`  {`
`  count++;`
`  }`
`  return count;`
`}`
	`volatile int buffer_full;`
	`int read_stream(void)`
	`{`
	`  int count = 0;`
	`  while (!buffer_full)`
	`  {`
	`  count++;`
	`  }`
	`  return count;`
	`}`

表 5-6 是编译器为第5-4 页的表 5-1 中的每个示例实现生成的机器代码的反汇编，其中，每个实现的 C 代码均使用选项 -O2 进行了编译。

表 5-6 非易失性和易失性缓冲区循环的反汇编

非易失性版本的缓冲区循环	易失性版本的缓冲区循环

`read_stream PROC`
`  LDR  r1, |L1.28|`
`  MOV  r0, #0`
`  LDR  r1, [r1, #0]`
`|L1.12|`
`  CMP  r1, #0`
`  ADDEQ  r0, r0, #1`
`  BEQ  |L1.12|  ; infinite loop`
`  BX  lr`
`  ENDP`
`|L1.28|`
`  DCD  ||.data||`
`  AREA ||.data||, DATA, ALIGN=2`
`buffer_full`
`  DCD  0x00000000`
	`read_stream PROC`
	`  LDR  r1, |L1.28|`
	`  MOV  r0, #0`
	`|L1.8|`
	`  LDR  r2, [r1, #0];  ; buffer_full`
	`  CMP  r2, #0`
	`  ADDEQ  r0, r0, #1`
	`  BEQ  |L1.8|`
	`  BX  lr`
	`  ENDP`
	`|L1.28|`
	`  DCD  ||.data||`
	`  AREA ||.data||, DATA, ALIGN=2`
	`  buffer_full`
	`  DCD  0x00000000`

在表 5-6 的非易失性版本的缓冲区循环的反汇编中，语句 `LDR r0, [r0, #0]` 在标记为 `|L1.8|` 的循环之外将 `buffer_full` 的值加载到寄存器 `r0` 中。因为未将 `buffer_full` 声明为 **volatile**，编译器假定其值不能在程序之外进行修改。鉴于 `buffer_full` 的值已读取到 `r0` 中，编译器在启用优化时将忽略重新加载变量，这是因为其值不能更改。得到的结果是标记为 `|L1.8|` 的无限循环。

与之相对的是，在易失性版本的缓冲区循环的反汇编中，编译器假定 `buffer_full` 的值可以在程序之外更改并且不执行优化。因此，`buffer_full` 的值将加载到标记为 `|L1.4|` 的循环之内的寄存器 `r0` 中。因此，循环 `|L1.4|` 可以在汇编代码中正确执行。

要避免在执行之外更改程序状态造成的优化问题，当变量的值会由执行所不知道的方式意外更改时，必须将变量声明为 **volatile**。在实际编程中，出现以下情况时必须将变量声明为 **volatile**：

- 访问内存映射的外围设备
- 多个线程之间共享全局变量
- 在中断例程中访问全局变量。

5.2 代码度量

代码度量提供了一种客观评估代码质量的方法。ARM 编译器、链接器和性能分析器提供了几种能生成简单代码度量和提高代码质量的功能。特别是可以：

- 测量代码和数据大小
- 生成静态调用图
- 测量堆栈使用
- 减少对象文件和库中的调试信息。

有关 ARM Profiler 的详细信息，请参阅 《ARM Profiler 用户指南》。

5.2.1 测量代码和数据大小

通过使用一系列选项可以测量应用程序的代码和数据大小。请参阅：

- *《编译器参考指南》*中第2-73 页的*--info=totals*
- *《实用程序指南》*中第2-30 页的*--info=topic[,topic,...]*
- *《链接器参考指南》*中第2-7 页的*--[no_]callgraph*
- *《链接器参考指南》*中第2-37 页的*--[no_]map*
- *《链接器参考指南》*中第2-56 页的*--[no_]symbols*
- *《链接器参考指南》*中第2-62 页的*--[no_]xref*。

5.2.2 测量堆栈使用

C 和 C++ 均大量使用堆栈。例如，使用堆栈存放：

- 函数的返回地址
- 由 AAPCS 确定的必须保留的寄存器
- 局部变量，包括局部数组、结构和 C++ 中的类。

一般情况下是无法自动测量堆栈的使用的。但是，可以手动估算堆栈的使用范围。这可以按以下多种方法进行：

- 与 `--callgraph` 链接以生成静态调用图。这样将显示所有函数的信息，包括堆栈使用。
- 与 `--info=stack` 或 `--info=summarystack` 链接以列出所有全局符号的堆栈使用量。
- 使用调试器在堆栈中最后的可用位置设置观察点，然后查看是否命中过该观察点。

- 使用调试器可以：
  1. 为堆栈分配大大超过您预期所需的空间。
  2. 使用已知值填充堆栈，例如，零或 0xDEADDEAD。
  3. 运行应用程序或其固定部分。在测试运行中使用尽可能多的堆栈。例如，请确保执行了代码中尽可能多的跳转，并在合适的位置生成中断，使其可包含在堆栈跟踪中。
  4. 应用程序执行完毕后，检查内存的堆栈区以查看被覆盖的已知值（零或 0xDEADDEAD）的数目。堆栈显示已使用堆栈部分的碎片以及剩余部分中的零值或 0xDEADDEAD 值。
  5. 计算已知条目的数量并乘以 8。这显示了堆栈在内存中的增长量（以字节计）。
- 对于 RVISS，请使用映射文件来定义不允许访问的内存区。将此内存区放置在内存中堆栈的正下方。如果堆栈溢出到禁止区，则会发生数据中止，这可以由调试器捕获。

请参阅：
- 第5-9 页的*测量代码和数据大小*
- *《链接器参考指南》*中第2-7 页的*--[no_]callgraph*

5.2.3 减少对象文件和库中的调试信息

这通常在减少对象文件和库中的调试信息量时有用。降低调试信息级别：

- 减少对象和库的大小，从而减少存储它们所需的磁盘空间量。
- 加快链接时间。在编译周期中，大多数链接时间消耗在读取所有调试段和消除重复上。
- 将最终映像的大小最小化。这有助于调试器快速加载和处理调试符号。

有多种方法可以减少每个源文件生成的调试信息量。例如，您可以：

- 避免在头文件中使用 `#define` 条件。这样做可能导致链接器比较难以删除重复的信息。
- 修改 C 或 C++ 源文件，以便按相同的顺序包含 (`#include`) 头文件。
- 将头文件信息分拆分为小块。也就是说，使用大量的小头文件，而不是使用少量的大头文件。这有助于链接器消除更多的公用块。
- 只有在真正需要时，才将头文件包括在 C 或 C++ 源文件中。

- 防止头文件中的多重包含。例如，如果已有头文件 foo.h，然后添加：

```
#ifndef foo_h
#define foo_h
...
// rest of header file as before
...
#endif /* foo_h */
```

  则可以使用编译器选项 --remarks 以发出有关无保护的头文件警告。

- 使用 --no_debug_macros 命令行选项编译代码，以放弃调试表中的预处理器宏定义。

请参阅：

- *《编译器参考指南》*中第2-37 页的*--debug_macros, --no_debug_macros*
- *《编译器参考指南》*中第2-108 页的*--remarks*。

5.3 函数

要使编译器能更有效地执行优化，通常而言，保持小且简单的函数是一种好方法。有多种方法可以实现此目标。例如，您可以：

- 将传递自/至函数的参数数目最小化
- 使用 __value_in_regs 通过寄存器从函数返回多个值
- 可能时，将函数限定为 __pure。

5.3.1 将参数传递开销最小化

有多种方法可以将参数传递到函数的开销最小化。例如：

- 如果每个参数的大小都是一个字或者更小，请确保函数的参数不超过四个。在 C++ 中，请确保非静态成员函数的参数不超过三个，因为有隐式的 this 指针参数，它通常在 R0 中传递。
- 如果函数需要四个以上参数，请确保其执行大量的工作，以便超过传递堆栈参数所需的开销。
- 将相关自变量放在结构中，并在任何函数调用中将指针传递到结构。这减少了参数数目并增加了可读性。
- 尽量减少 **long long** 参数的数目，因为这种参数接受两个需要在偶数寄存器索引上对齐的自变量。
- 如果启用了软件浮点，则尽量减少 **double** 参数的数目。
- 避免具有可变参数数目的函数。使用可变数目自变量的函数可以在堆栈中有效地传递其所有自变量。

5.3.2 __value_in_regs

在 C 和 C++ 中，一种从一个函数返回多个值的方法是使用结构。通常，结构在堆栈上返回，需要承担所有产生的相关开销。

为了减少内存通信并减少代码大小，可以借助编译器通过寄存器从函数返回多个值。通过 __value_in_regs 限定函数，最多可以从 **struct** 中的函数返回四个字。例如：

```
typedef struct s_coord { int x; int y; } coord;
coord reflect(int x1, int y1) __value_in_regs;
```

在任何需要从一个函数返回多个值的位置，都可以使用 __value_in_regs。示例包括：

- 从 C 和 C++ 函数返回多个值
- 从嵌入式汇编语言函数返回多个值
- 进行超级用户调用
- 重新实现 __user_initial_stackheap。

有关 __value_in_regs 的详细信息，请参阅 《编译器参考指南》中第4-20 页的 *__value_in_regs*。

5.3.3 __pure

纯函数是在使用相同参数进行调用时始终返回相同结果的函数。

根据定义，对于任何特定的纯函数调用，只需求一次值就足够了。因为只要是相同的调用，函数调用的结果必然相同，代码中所有的后续函数调用都可以替换为第一次调用的结果。

若要向编译器指示函数为纯函数，请将函数声明为 __pure。

在表 5-7 的两个示例例程中说明了 __pure 关键字的用法。两个例程均调用函数 fact 来计算 n! 和 n! 的总和。fact 函数仅根据其输入自变量 n 来计算 n!。因此，fact 是纯函数。

第一个例程显示了简单的函数 fact 的执行，其中未将 fact 声明为 __pure。在第二个执行中，将函数 fact 限定为 __pure 来向编译器指示其为纯函数。

表 5-7 纯函数和非纯函数的 C 代码

未声明为 __pure 的纯函数	声明为 __pure 的纯函数

`int fact(int n)`
`{`
`  int f = 1;`
`  while (n > 0)`
`  f *= n--;`
`  return f;`
`}`
`int foo(int n)`
`{`
`  return fact(n)+fact(n);`
`}`
	`int fact(int n) __pure`
	`{`
	`  int f = 1;`
	`  while (n > 0)`
	`  f *= n--;`
	`  return f;`
	`}`
	`int foo(int n)`
	`{`
	`  return fact(n)+fact(n);`
	`}`

表 5-8 是编译器为第5-13 页的表 5-7 中的每个示例实现生成的机器代码的反汇编，其中，每个实现的 C 代码均使用选项 -O2 进行了编译。

表 5-8 纯函数和非纯函数的反汇编

未声明为 __pure 的纯函数	声明为 __pure 的纯函数

fact PROC
  ...
foo PROC
  MOV r3, r0
  PUSH {lr}
  BL fact
  MOV r2, r0
  MOV r0, r3
  BL fact
  ADD r0, r0, r2
  POP {pc}
  ENDP
	fact PROC
	  ...
	foo PROC
	  PUSH {lr}
	  BL fact
	  LSL r0,r0,#1
	  POP {pc}
	  ENDP

在表 4-8 中，未将 fact 限定为 __pure 的情况下，函数 foo 的反汇编调用两次函数 fact，这是因为编译器不知道可对该函数执行 CSE 操作。与之相对，在表 4-8 中，将 fact 限定为 __pure 的 foo 的反汇编只调用一次 （而不是两次） fact，这是因为编译器可以在添加 fact(n) + fact(n) 时执行 CSE 操作。

根据定义，纯函数没有副作用。例如，纯函数无法使用全局变量或间接通过指针来读取或写入全局状态，因为访问全局状态会与函数在两次调用相同参数时每次必须返回相同值的规则冲突。因此，在程序中必须谨慎地使用 __pure。然而，在函数可以声明为 __pure 的地方，编译器通常可以执行强力优化，如 CSE。

有关纯函数的详细信息，请参阅 《编译器参考指南》中第4-13 页的*__pure*。

5.3.4 放置 ARM 函数限定符

许多 ARM 关键字扩展可以修改函数的行为或调用顺序。例如， __pure、 __irq、 __swi、 __swi_indirect、 __softfp 和 __value_in_regs 均具有这种作用。

这些函数修饰符均具有公共语法。函数修饰符 （如 __pure）可以按如下方式限定函数声明：

- 在函数声明之前。例如：

  ```
  __pure int foo(int);
  ```

- 在参数列表的结束括号之后。例如：

```
int foo(int) __pure;
```

对于简单函数声明，每种语法都是明确的。但是，对于返回类型或自变量为函数指针的函数，前缀语法是不精确的。例如，下面的函数返回函数指针，但不清楚 __pure 是修改函数自身还是其返回的指针类型：

```
__pure int (*foo(int)) (int); /* declares 'foo' as a (pure?) function that
                                 returns a pointer to a (pure?) function.
                                 It is ambiguous which of the two function
                                 types is pure. */
```

实际上，在 foo 声明之前放置一个 __pure 关键字，会修改 foo 自身和 foo 返回的函数指针类型。

与之相对的是后缀语法，借助它，可以在声明其参数和返回类型是函数指针的函数时，明确区分 __pure 是应用到参数、返回类型还是基本函数。例如：

```
int (*foo1(int) __pure) (int);        /* foo1 is a pure function returning
                                         a pointer to a normal function */
int (*foo2(int)) (int) __pure;        /* foo2 is a function returning
                                         a pointer to a pure function */
int (*foo3(int) __pure) (int) __pure; /* foo3 is a pure function returning
                                         a pointer to a pure function */
```

在本示例中：

- foo1 和 foo3 修改自身
- foo2 和 foo3 返回指向修改过的函数的指针
- 函数 foo3 和 foo 相同。

因为后缀语法比前缀语法更精确，所以建议在使用 ARM 函数修饰符限定函数时，尽可能使用后缀语法。

5.4 函数内联

函数内联提供了代码大小和性能之间的折衷方案。缺省情况下，编译器自行确定是否内联代码。作为通则，编译器以生成最小大小的代码为出发点对内联做出合理决策。这是因为代码大小对于嵌入式系统非常重要。

在大多数情况下，最好由编译器确定是否内联特定函数。不过，通过使用相应的 inline 关键字，可以提示编译器需要内联某个函数。编译器还提供了一系列其他功能，用于修改其与内联相关的行为。在确定是否使用这些功能时，必须考虑几个因素，或从更一般的意义上来说，是否需要内联函数。

用 `__inline`、`inline` 或 `__forceinline` 限定的函数称为内联函数。在 C++ 中，在类、结构或联合内部定义的成员函数也是内联函数。

注意

请注意，由性能分析主导的优化可影响函数内联。请参阅 *《编译器参考指南》*中第2-104 页的*--profile=filename*。

5.4.1 编译器如何确定内联

如果启用了内联，编译器将使用复杂决策树来确定何时内联函数。编译器使用以下简化算法确定是否要内联函数：

1. 如果函数是使用 `__forceinline` 限定的，则尽可能内联函数。
2. 如果函数是使用 `__inline` 限定的，并且选择了选项 `--forceinline`，则尽可能内联函数。

   如果函数是使用 `__inline` 限定的，但没有选择选项 `--forceinline`，则在适合的情况下内联函数。
3. 如果优化级别为 `-O2` 或更高，或者选择了 `--autoinline`，则在适合的情况下尽可能内联函数。

在确定内联函数是否实际可行时，编译器需要考虑其他几个标准，包括选择 `-Ospace` 还是 `-Otime`。选择 `-Otime` 可以增加函数内联的可能性。请参阅第5-17 页的*什么时候才适合编译器执行内联？*。

您无法覆盖编译器做出的关于何时适合内联函数的决策。例如，如果编译器认为不合理，您无法强制函数内联。

5.4.2 什么时候才适合编译器执行内联？

编译器根据一系列条件自行确定什么时候适合内联函数，包括：

- 函数的大小以及调用的次数
- 当前优化级别
- 针对速度优化 (-Otime) 还是针对大小优化 (-Ospace)
- 函数是否具有外部或静态链接
- 函数的参数个数
- 是否使用函数的返回值

最后，即使使用 __forceinline 限定了函数，编译器也可以决定不内联函数。作为通则：

- 较小的函数内联的机会较大
- 使用 -Otime 编译会增加函数内联的可能性
- 大函数通常不内联，因为这样做会影响代码密度和性能。

5.4.3 管理内联

使用 __forceinline 关键字可以强制编译器尝试内联函数。如果不会导致出现问题，编译器将函数内联。例如，递归函数仅内联到本身一次。若要强制编译器尝试内联所有标记为 __inline 的函数，请使用 --forceinline 命令行选项编译代码。

在优化的最高级别 （-O2 和 -O3），即使您没有明确进行提示，编译器也能够在合理的情况下自动内联函数。请参阅第5-21 页的*将函数标记为静态*。

使用 --no_autoinline 和 --autoinline 命令行选项，可以在最高优化级别控制函数的自动内联。一般情况下，启用自动内联时，编译器在合理的情况下内联所有可以内联的内容。如果禁用了自动内联，则只有标记为 __inline 的函数才会进行内联。

使用 --no_inline 和 --inline 关键字可以控制是否执行内联。缺省情况下，函数内联是启用的。如果使用 --no_inline 命令行选项禁用函数内联，则编译器仅尝试内联使用 __forceinline 显式限定的函数。

请参阅：

- *《编译器参考指南》*中第2-16 页的*--autoinline*, *--no_autoinline*
- *《编译器参考指南》*中第2-58 页的*--forceinline*
- *《编译器参考指南》*中第2-73 页的*--inline*, *--no_inline*
- 《编译器参考指南》中第4-6 页的*__forceinline*

- 《编译器参考指南》中第4-9 页的*__inline*

5.4.4 自动内联

在 -O2 和 -O3 优化级别，编译器将尝试内联对定义的函数的调用，而不是内联函数。这对于静态函数最有用，因为如果在使用静态函数的位置都进行内联，则不需要外部副本。如果非内联函数不在定义它们的转换单元之外使用，最好将所有这些非内联函数标记为静态函数。转换单元是对源文件与所有通过 #include 指令包含的头文件和源文件进行预处理的结果。通常，您不会希望将非内联函数的定义放在头文件中。

如果要使用 --multifile 选项 （-O3 级别缺省启用此选项）或 --ltcg 进行编译，编译器可以自动内联对其他转换单元内定义的函数的调用。

如果使用 --multifile，则两个转换单元必须在同一次编译器调用中进行编译。如果使用 --ltcg，只需将它们链接到一起。

--no_inline 用于禁用自动内联。

5.4.5 C++、 C90、 C99 和 GNU C90 编译器模式之间的行为差异

在某些方面，内联的效果是有差异的，具体取决于编译器编译时所针对的语言。

__forceinline 的行为类似于 __inline，只是编译器执行内联的力度更大。

C++ 和 C90 模式

C90 中没有 **inline** 关键字。

C90 中的 **__inline** 以及 C++ 中的 **__inline** 和 **inline** 的作用是一样的。

在声明要内联的 extern 函数时，必须在每个需要使用该函数的转换单元中对其进行定义。必须确保在每个转换单元中使用相同的定义。

即使该函数有外部链接，也必须在每个转换单元中定义它。

如果一个内联函数由多个文件使用，通常将其定义放在头文件中。

不建议将非内联函数的定义放在头文件中，因为这样可能导致在每个转换单元中都创建一个单独的函数。如果非内联函数是 **extern** 函数，在链接时就会出现重复的符号。如果非内联函数是 **static** 函数，就会出现多余的重复代码。

在 C++ 结构、类或联合的声明中定义的成员函数是隐式内联函数。这些函数被视为以 **inline** 或 **__inline** 关键字声明的函数。

内联函数有 **extern** 链接，除非它们显式声明为 **static** 函数。如果某个内联函数声明为静态函数，则该函数的外部副本对于其转换单元必须是唯一的，所以将内联函数声明为静态函数可能导致多余的重复代码。

如果编译器无法内联函数，或者确定不内联函数，则会生成对函数外部副本的常规调用。

如果在使用函数的每个转换单元内都定义该函数，则编译器不需要生成所有 **extern** 内联函数的外部副本。如果编译器要生成 **extern** 内联函数的外部副本，它使用 “公共组”，以便链接器删除重复代码，最多为不同对象文件的同一外部函数保留一个副本。（请参阅 《链接器用户指南》中第3-10 页的 *公共组或节删除*。）

C99 模式

对于含有外部链接的 C99 内联函数，其适用规则不同于 C++。C99 区分内联定义和外部定义。在给定的定义内联函数的转换单元中，如果内联函数从不以 **extern** 声明而始终以 **inline** 声明，则这种定义是内联定义。否则就是外部定义。即使使用 `--no_inline`，这些内联定义也不用于生成外部副本。

内联函数的每次使用，可能通过同一转换单元内的定义（可以是内联定义或外部定义）进行内联，也可能成为外部定义调用。如果使用内联函数，则必须在某个转换单元中有（且只有）一个外部定义。如果使用外部函数，这一规则同样适用。实际上，如果内联函数的所有使用都进行内联，则在缺少外部定义时不会出错。如果使用 `--no_inline`，则只使用外部定义。

通常，应使用 **inline** 而非 **extern** 将带外部链接的内联函数放在头文件中作为内联定义。源文件中也有外部定义。例如：

示例 5-1 C99 中的函数内联

```
/* example_header.h */
inline int my_function (int i)
{
   return i + 42; // inline definition
}

/* file1.c */
#include  "example_header.h"
   ... // uses of my_function()
```

```
/* file2.c */
#include  "example_header.h"
   ... // uses of my_function()

/* myfile.c */
#include  "example_header.h"
extern inline int my_function(int); // causes external definition.
```

在 C++ 中通常也采用这种策略，但 C++ 中没有专门的外部定义，也不需要外部定义。

在不同的转换单元中，内联函数的定义可以不同。但是，在一般的使用情况下，如 第5-19 页的*C99 中的函数内联*，它们是相同的。

在使用 `--multifile` 或 `--ltcg` 进行编译时，在一个转换单元中的调用可能使用另一个转换单元中的外部定义进行内联。

C99 对内联定义设了一些限制：不能定义可修改的局部静态对象，不能引用带静态链接的标识符。

与所有其他模式一样，在 C99 模式中，**`__inline`** 和 **`inline`** 的作用是相同的。

带静态链接的内联函数在 C99 中的行为与在 C++ 中一样。

GNU C90 模式

GNU C90 内联规则与其他编译器模式中的规则不同。请参阅 GNU 文档，地址是 `http://gcc.gnu.org`。

5.4.6 链接器内联

链接器可以将对某些很短的函数的调用替换为函数体。请参阅 *《链接器参考指南》* 中 第2-28 页的*`--[no_]inline`*。

5.4.7 调试数据和 --no_inline 及 --inline 命令行选项

为内联函数的使用生成的调试视图通常都很好。但在某些时候，不进行函数内联很有用，这是因为在某些情况下，如果不内联函数，调试会更清晰。可以使用 `--no_inline` 和 `--inline` 命令行选项来启用和禁用函数内联。请参阅 *《编译器参考指南》* 中第2-73 页的*`--inline`*, *`--no_inline`*。

5.4.8 将函数标记为静态

在优化级别 -O2 和 -O3，即使未将函数声明为 __inline 或 **inline**，编译器也可以在适合的情况下自动内联函数。

注意

要控制更高优化级别上的函数自动内联，请使用 --no_autoinline 和 --autoinline 命令行选项。

除非将函数显式声明为 **static**（或 __inline），否则从其他模块调用该函数时，编译器在对象文件中保留其外部版本。链接器无法将未使用的外部函数从对象文件中删除，除非使用以下方法之一将它们放置在自身所在的代码段中：

- *《编译器参考指南》*中第2-115 页的*--split_sections*
- *《编译器参考指南》*中第4-50 页的 *__attribute__((section("name")))*
- *《编译器参考指南》*中第4-56 页的*#pragma arm section [section_sort_list]*
- 链接器反馈

如果未将从不从模块外部调用的函数声明为 **static**，则会对代码造成不利影响。特别是可能会造成：

- 较大的代码大小，因为函数的外部版本是在映像中保存的。

  函数自动内联时，函数的内联版本和外部版本均可能在最终映像中终止，除非将函数声明为 **static**。这可能会增加代码大小。

- 不必要的复杂调试视图，因为要显示函数的外联版本和内联版本。

  在代码中保留函数的内联和外联副本，有时在调试视图中设置断点或单步执行时可能会造成混淆。调试器需要在其交叉存取视图中显示内联和外部版本，因此您可以在单步执行内联或外部版本时查看具体情况。

因为可能出现这些问题，如果确定不会从其他模块调用非内联函数，请将该函数声明为 **static**。

5.4.9 在 ROM 映像中为内联函数设置断点

为内联函数设置断点时， RealView Debugger 将尝试为函数的每个内联实例设置断点。如果要使用 RealView ICE 来调试 ROM 中的映像，并且内联实例的数量大于可用硬件断点的数量，则调试器可能无法设置额外的断点。在这种情况下，调试器将报告错误。

请参阅：

- *《编译器参考指南》*中第2-16 页的*--autoinline*, *--no_autoinline*
- *《编译器参考指南》*中第2-58 页的*--forceinline*
- *《编译器参考指南》*中第2-73 页的*--inline*, *--no_inline*
- *《编译器参考指南》*中第4-6 页的*__forceinline*
- *《编译器参考指南》*中第4-9 页的*__inline*

5.5 对齐数据

各种 C 语言数据类型都在特定字节边界上对齐以最大限度地利用存储空间，并为使用 ARM 指令集的操作提供快速高效的内存访问。例如，当对象存储在可被四整除的地址处时，ARM 体系结构只需使用一个指令即可访问一个四字节变量，因此，四字节对象位于四字节边界上。

缺省情况下，编译器按表 5-9 中所示来存储数据对象。

表 5-9 编译器按字节对齐存储数据对象

类型	字节	对齐
`char`	1	位于字节地址处。
`short`	2	位于可被 2 整除的任何地址处。
`float`, `int`, `long`, `pointer`	4	位于可被 4 整除的某一地址处。
`long long` `double`	8	位于可被 4 整除的某一地址处。

当编译器将变量放置在物理内存地址处时，数据对齐将变得互相关联。例如，在以下结构中，`bmem` 和 `cmem` 之间需要三字节的空隙。

```
struct example_st {
  int amem;
  char bmem;
  int cmem;
};
```

ARM 和 Thumb 处理器用于高效访问自然对齐的数据，即其地址是四的倍数的双字、其地址是四的倍数的字、其地址是二的倍数的半字以及其地址是任何字节地址的单字节。此类数据位于其自然大小边界。

5.5.1 数据对齐的类型

所有的内存数据访问都可以分为以下类别：

- 自然对齐，例如，在位于 0x1000 的字边界上。ARM 编译器通常对齐变量并填充结构，使得可以使用 LDR 和 STR 指令有效访问这些项。
- 已知但并非自然对齐，例如，位于地址 0x1001 的字。此类对齐通常出现在压缩结构以删除不必要的填充时。在 C 和 C++ 中，__packed 限定符或 #pragma pack(*n*) 编译指令用于表示结构已压缩。
- 未知对齐，例如，位于任意地址的字。此对齐类型通常出现在定义可以指向位于任何地址的字的指针时。在 C 和 C++ 中，__packed 限定符或 #pragma pack(*n*) 编译指令用于表示指针可以访问位于非自然对齐边界的字。

有关 __packed 限定符、压缩结构和未对齐指针的详细信息，请参阅第5-25 页的 *__packed 限定符和未对齐的数据访问*。

有关 #pragma pack(*n*) 的信息，请参阅 *《编译器参考指南》*中第4-65 页的 *#pragma pack(n)*。

5.5.2 未对齐的数据访问

访问内存中未对齐的数据是必要的，例如，从 CISC 体系结构 （指令可直接访问内存中的未对齐数据）移植旧代码时。

在 ARMv4 和 ARMv5 体系结构中，以及在 ARMv6 体系结构 （视配置方式而定）中，访问内存中的未对齐数据时需要很小心，以免出现意外的结果。例如，使用常规指针读取 C 或 C++ 源代码中的字时，ARM 编译器将生成使用 LDR 指令读取字的汇编语言代码。这在地址是四的倍数时工作正常，例如，当其位于字的边界时。但是，如果地址不是四的倍数，则 LDR 返回循环结果而不是执行真正的未对齐字加载。通常此循环不是程序员预期的结果。

ARMv6 和更高版本的体系结构完全支持未对齐访问。

5.5.3 __packed 限定符和未对齐的数据访问

__packed 限定符将任意有效类型的对齐设置为 1。这样可以使用未对齐访问读取或写入压缩类型的对象。

可以压缩的对象示例包括：

- 结构
- 联合
- 指针。

有关 __packed 限定符的详细信息，请参阅 *《编译器参考指南》*中第4-11 页的 *__packed*。

结构中的未对齐字段

为提高效率，结构中的字段位于其自然大小边界。这意味着编译器通常需要在字段之间插入填充以确保对齐。

当空间紧张时，可以使用 __packed 限定符创建字段之间没有填充的结构。可以通过两种方式压缩结构：

- 可以将整个 **struct** 声明为 __packed。例如：

  ```
  __packed struct mystruct
  {
      char c;
      short s;
  } // not recommended
  ```

  结构的每个字段都继承 __packed 限定符。

  将整个 **struct** 声明为 __packed 通常会对代码大小和性能造成不利影响。请参阅第5-27 页的*__packed 结构与单个 __packed 字段*。

- 可以将 **struct** 中单个未对齐的字段声明为 __packed。例如：

  ```
  struct mystruct
  {
      char c;
      __packed short s; // recommended
  }
  ```

  这是推荐的压缩结构的方法。请参阅第5-27 页的*__packed 结构与单个 __packed 字段*。

——注意——

对于联合，适用相同的原则。您可以将整个联合声明为 __packed，也可以使用 __packed 属性标识内存中未对齐联合的组件。

读取或写入使用 __packed 限定的结构需要进行未对齐访问，可能因此影响性能。请参阅第5-27 页的*__packed 结构与单个 __packed 字段*。

未对齐的指针

缺省情况下，ARM 编译器需要常规 C 指针指向内存中对齐的字，因为编译器可以生成更高效的代码。

如果要定义可以指向位于任何地址的字的指针，则必须在定义指针时使用 __packed 限定符进行指定。例如：

```
__packed int *pi; // pointer to unaligned int
```

将指针声明为 __packed 时，ARM 编译器将生成正确访问指针的解除引用值的代码，不论其是否对齐。生成的代码包括字节访问序列，或者依赖对齐的可变移位和屏蔽指令，而不是简单的 LDR 指令。因此，将指针声明为 __packed 会对性能和代码大小造成不利影响。

用于访问半字的未对齐 LDR 指令

在某些情况下，编译器可能会有意生成未对齐的 LDR 指令。特别需要指出的是，即使体系结构支持专用的半字加载指令，编译器会也会生成此类指令以从内存中加载半字。

例如，要访问 __packed 结构中未对齐的 **short**，编译器可能会将所需的半字加载到寄存器的上半部分，然后向下移动到下半部分。此运算只需要访问一次内存，而使用 LDRB 指令执行相同的运算需要访问两次内存，还要加上要合并两个字节的指令。

5.5.4 __packed 结构与单个 __packed 字段

优化已压缩的 **struct** 时，编译器尝试推算每个字段的对齐以改善访问。但是，编译器并非总是可以推算 __packed **struct** 中每个字段的对齐。与之相对的是，将 struct 中的单个字段声明为 __packed 时，可以确保对 **struct** 中自然对齐成员的快速访问。因此，需要使用压缩结构时，建议您始终压缩结构中的单个字段，而不是整个结构本身。

注意

将 **struct** 的单个未对齐字段声明为 __packed 还具有其他的优点：程序员可以更清楚的看出 **struct** 的哪些字段没有对齐。

有关不压缩 **struct**、压缩整个 **struct** 和压缩 **struct** 的单个字段三者之间的差别，将使用表 5-10 中显示的三个 **struct** 的执行来加以说明。

在第一个执行中， **struct** 没有压缩。在第二个执行中，将整个结构 mystruct 限定为 __packed。在第三个执行中，从 mystruct 结构中删除了 __packed 属性，并且将单个未对齐字段声明为 __packed。

表 5-10 未压缩 struct、压缩的 struct 和具有单独压缩字段的 struct 的 C 代码

未压缩 struct	__packed struct	__packed 字段

struct foo
{
    char one;
    short two;
    char three;
    int four;
} c;
	__packed struct foo
	{
	    char one;
	    short two;
	    char three;
	    int four;
	} c;
		struct foo
		{
		    char one;
		    __packed short two;
		    char three;
		    int four;
		} c;

第5-28 页的表 5-11 是编译器为表 5-10 中的每个示例实现生成的机器代码的反汇编，其中，每个实现的 C 代码均使用选项 -O2 进行了编译。

注意

-Ospace 和 -Otime 编译器选项控制对未对齐元素的访问是内联进行还是通过函数调用。使用 -Otime 将导致内联未对齐访问，而使用 -Ospace 则导致调用函数来进行未对齐访问。

表 5-11 未压缩 struct、压缩的 struct 和具有单独压缩字段的 struct 的反汇编

未压缩 struct	__packed struct	__packed 字段

; r0 contains address of c
; char one
LDRB r1, [r0, #0]
; short two
LDRSH r2, [r0, #2]
; char three
LDRB r3, [r0, #4]
; int four
LDR r12, [r0, #8]
	; r0 contains address of c
	; char one
	LDRB r1, [r0, #0]
	; short two
	LDRB r2, [r0, #1]
	LDRSB r12, [r0, #2]
	ORR r2, r12, r2, LSL #8
	; char three
	LDRB r3, [r0, #3]
	; int four
	ADD r0, r0, #4
	BL __aeabi_uread4
		; r0 contains address of c
		; char one
		LDRB r1, [r0, #0]
		; short two
		LDRB r2, [r0, #1]
		LDRSB r12, [r0, #2]
		ORR r2, r12, r2, LSL #8
		; char three
		LDRB r3, [r0, #3]
		; int four
		LDR r12, [r0, #4]

在表 5-11 的未压缩 **struct** 的反汇编中，编译器始终访问位于对齐字或半字地址上的数据。编译器可以执行此操作是因为填充了 **struct**，因此 **struct** 的每个成员均位于自然大小边界上。

在表 5-11 中的 __packed **struct** 反汇编中，缺省情况下，字段 one 和 three 在其自然大小边界上对齐，因此编译器可以进行对齐访问。对可以将其标识为对齐的字段，编译器始终执行对齐字或半字的访问。对于未对齐字段 two，编译器使用多重对齐内存访问 (LDR/STR/LDM/STM)，与固定移位和屏蔽组合在一起，用于访问内存中的正确字节。编译器调用 AEABI 运行时例程 __aeabi_uread4 以读取未知对齐的无符号字，从而访问字段 four，原因为无法确定该字段是否位于其自然大小边界上。

在表 5-11 中具有单独压缩字段的 **struct** 的反汇编中，对字段 one、two 和 three 的访问方式，与在将整个 **struct** 限定为 __packed 时的访问方式相同。但是，与压缩整个 **struct** 的情况相反，编译器对字段 four 进行字对齐访问，因为结构中的 __packed short 的存在可以帮助编译器确定字段 four 位于其自然大小边界上。

5.6 使用浮点算法

ARM 编译器在软件和硬件方面提供了多种管理浮点算法的功能。例如，您可以指定浮点的软件或硬件支持，特别是硬件体系结构，以及符合 IEEE 浮点标准的级别。

对浮点选项的选择将决定浮点性能、系统开销和系统灵活性之间的折衷方案。要在性能、开销和灵活性之间获得最佳折衷方案，您需要对浮点选项进行合理的选择。

5.6.1 浮点运算的支持

ARM 处理器内核不包含浮点硬件。必须使用以下两种方法之一，另行提供对浮点算法的支持：

- 在软件中，使用浮点库 `fplib`。此库提供了执行浮点运算可以调用的函数，无需额外的硬件。请参阅 *《库指南》*中第4-2 页的*软件浮点库 fplib*。
- 在硬件中，使用含 VFP 硬件协处理器的 ARM 处理器内核来进行所需的浮点运算。VFP 是执行 IEEE 浮点的协处理器体系结构，支持单精度和双精度，但不支持扩展精度。

  注意

  在实际编程中， VFP 中的浮点运算实际是组合使用硬件 （执行常见的情况）和软件 （处理不常见的情况和导致异常的情况）执行的。请参阅第5-32 页的*VFP 支持*。

示例 5-2 是一个用 C 执行浮点算法的函数，用以说明浮点算法的软件和硬件支持的不同。

示例 5-2 浮点运算

```
float foo(float num1, float num2)
{
    float temp, temp2;
    temp = num1 + num2;
    temp2 = num2 * num2;
    return temp2-temp;
}
```

如果使用命令行选项 --cpu 5TE --fpu softvfp 编译第5-29 页的示例 5-2 的 C 代码，则编译器生成的机器代码的反汇编如示例 5-3 所示。在本示例中，在软件中通过调用库例程 （如 __aeabi_fmul）来执行浮点算法。

示例 5-3 软件中对浮点运算的支持

```
||foo|| PROC
    PUSH     {r4-r6, lr}
    MOV      r4, r1
    BL       __aeabi_fadd
    MOV      r5, r0
    MOV      r1, r4
    MOV      r0, r4
    BL       __aeabi_fmul
    MOV      r1, r5
    POP      {r4-r6, lr}
    B        __aeabi_fsub
    ENDP
```

如果使用命令行选项 --fpu vfp 编译第5-29 页的示例 5-2 的 C 代码，则编译器生成的机器代码的反汇编如示例 5-4 所示。在本示例中，在硬件中通过浮点算法指令 （如 VMUL.F32）来执行浮点算法。

示例 5-4 硬件中对浮点运算的支持

```
||foo|| PROC
    VADD.F32 s2, s0, s1
    VMUL.F32 s0, s1, s1
    VSUB.F32 s0, s0, s2
    BX       lr
    ENDP
```

在实际编程中，使用硬件支持浮点算法的代码更为紧凑，并提供比在软件中执行浮点算法的代码更佳的性能。但是，浮点算法的硬件支持需要 VFP 协处理器。

缺省情况下，如果有 VFP 协处理器，则会生成 VFP 指令。如果没有 VFP 协处理器，则编译器会生成调用软件浮点库 fplib 的代码，用于执行浮点运算。 fplib 是 C 库 RealView Development Suite 标准分发的组成部分。

5.6.2 VFP 体系结构

VFP 是提供单精度和双精度运算的浮点体系结构。许多运算可以标量格式或向量格式进行。支持多种版本的体系结构，包括：

- VFPv2，实现该体系结构的版本包括：
  - VFP10 修订版 1，由 ARM10200E 提供
  - VFP9-S，作为 ARM926E/946E/966E 的单独许可选项提供
  - VFP11，在 ARM1136JF-S、 ARM1176JZF-S 和 ARM11 MPCore 中提供。
- VFPv3，在 ARM 体系结构 v7 和更高的版本上实现，例如 Cortex-A8。VFPv3 向后兼容 VFPv2，但它不能捕获浮点异常。它不需要软件支持代码。VFPv3 具有 32 位双精度寄存器。
- VFPv3，可选择以半精度扩展方式加以扩展。这些扩展提供了一些转换函数，可在半精度浮点数和单精度浮点数之间进行双向转换。通过任何支持单精度浮点数的高级 SIMD 和 VFP 实现，都可以实现这些扩展。
- VFPv3-D16 是一种支持 16 位双精度寄存器的 VFPv3 实现。它在 ARM 体系结构 v7 处理器上实现，该处理器支持不含 NEON 的 VFP。
- VFPv3U 是可以捕获浮点异常的 VFPv3 实现。它需要软件支持代码。

——注意——

VFP 体系结构的特别执行可以提供其他特定于执行的功能。例如， VFP 协处理器硬件可能包括用于说明异常条件的附加寄存器。此附加功能称为 *子体系结构* 功能。有关子体系结构功能的详细信息，请参阅 《ARM 应用程序说明 133 - 使用带 RVDS 的 VFP》(*ARM Application Note 133 - Using VFP with RVDS*)。此应用程序说明位于 RealView Development Suite 分发 `Examples` 目录的 `vfpsupport` 子目录中，其路径为 *`install_directory`*`\RVDS\Examples\...\vfpsupport`。

5.6.3 VFP 支持

ARM VFP 协处理器进行了优化，以便在硬件中处理明确定义的浮点代码。硬件不处理极少出现的算法运算或太复杂的算法运算，而是由软件来处理这些情况。此方法将所需协处理器的数量最小化并降低开销。

为处理 VFP 硬件所无法处理的情况而提供的代码称为 VFP 支持代码。当 VFP 硬件无法直接处理某种情况时，它将此情况返回到 VFP 支持代码供进一步处理。例如，可能调用 VFP 支持代码来处理以下任意内容：

- 涉及 NaN 的浮点运算
- 涉及非正规数的浮点运算。
- 浮点溢出
- 浮点下溢
- 不精确结果
- 除零错误
- 无效运算。

支持代码到位后， VFP 将支持完全 IEEE 754 兼容的浮点模型。

使用 VFP 支持

为方便起见， RVCT 安装附带了可以在系统中使用的 VFP 支持代码的执行。支持代码由以下部分组成：

- 用于仿真硬件返回的 VFP 运算的库 `vfpsupport.l` 和 `vfpsupport.b`。

  这些文件位于 RVCT 安装的 \lib\armlib 子目录下。

- C 源代码和汇编语言源代码执行顶级、次级和用户级中断处理程序。

  这些文件位于 RealView Development Suite 分发版本的 `Examples` 目录的 `vfpsupport` 子目录中，其路径为 *`install_directory`*`\RVDS\Examples\...\vfpsupport`。

  可能需要对这些文件进行修改以将 VFP 支持与操作系统集成。

- 用于访问 VFP 协处理器子体系结构功能的 C 源代码和汇编语言源代码。

  这些文件在 RealView Development Suite 分发的 `Examples` 目录的 `vfpsupport` 子目录中，其路径为 *`install_directory`*`\RVDS\Examples\...\vfpsupport`。

VFP 协处理器返回指令时，向处理器发出 “未定义的指令”异常，并通过 “未定义的指令”向量输入 VFP 支持代码。顶级和次级中断处理程序执行某种初始信号处理，例如，确保异常不是由非法指令引起的。用户级的中断处理程序随后调用库 vfpsupport.l 或 vfpsupport.b 中的相应库函数来仿真软件中的 VFP 运算。

——注意——

以下情况无需使用 VFP 支持代码：

- 无需捕获不常见或异常情况时
- VFP 协处理器运行在 RunFast 模式下时
- 硬件协处理器是基于 VFPv3 的系统时。

有关使用 RVCT 安装附带的 VFP 支持代码的详细信息，请参阅 *《ARM 应用程序注释 133 - 使用带 RVDS 的 VFP》*。此应用程序说明在 RealView Development Suite 分发 Examples 目录的 vfpsupport 子目录中，其路径为 *install_directory*\RVDS\Examples\...\vfpsupport。

5.6.4 半精度浮点数支持

半精度浮点数是作为 VFPv3 体系结构的可选扩展提供的。如果 VFPv3 协处理器不可用，或者使用了 VFPv3 处理器但没有该扩展，则通过浮点库 fplib 来支持半精度浮点数。

只有当指定 fp16_format 命令行选项并选择半精度浮点数时，才能使用半精度浮点数。请参阅 *《编译器参考指南》*中第2-59 页的*--fp16_format=format*。

可用的半精度浮点格式有 ieee 和 alternative。在这两种格式中，16 位数的基本布局是相同的。请参阅图 5-1。

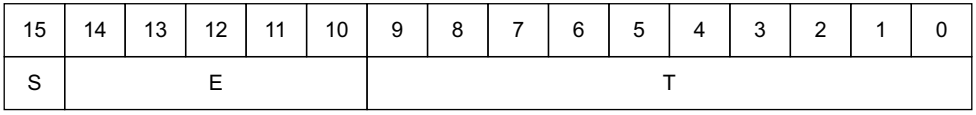

图 5-1 半精度浮点格式

其中：

```
S (bit[15]):      Sign bit
E (bits[14:10]):  Biased exponent
T (bits[9:0]):    Mantissa.
```

这些字段的含义取决于所选的格式。

IEEE 半精度

```
IF E==31:
   IF T==0: Value = Signed infinity
   IF T!=0: Value = Nan
             T[9] determines Quiet or Signalling:
                  0: Quiet NaN
                  1: Signalling NaN
IF 0<E<31:
   Value = (-1)^S x 2^(E-15) x (1+2^-10 T)

IF E==0:
   IF T==0: Value = Signed zero
   IF T!=0: Value = (-1)^S x 2^(-14) x (0+2^-10 T)
```

Alternative 半精度

```
IF 0<E<32:
   Value = (-1)^S x 2^(E-15) x (1+2^-10 T)

IF E==0:
   IF T==0: Value = Signed zero
   IF T!=0: Value = (-1)^S x 2^(-14) x (0+2^-10 T)
```

使用限制

在使用 __fp16 类型时，有以下限制：

- 在 C 或 C++ 表达式中使用时，__fp16 类型会提升为单精度。如果某个操作数需要，还可能继续提升为双精度。
- 单精度值可转换为 __fp16。双精度值先转换为单精度，再转换为 __fp16，此过程可能涉及两次舍入。这反映了 ARM 体系结构缺少双精度到 16 位的直接转换。
- 如果使用 fpmode=fast，半精度浮点格式与其他格式之间的转换不会引发浮点异常。
- 函数形参不能是 __fp16 类型。但是，指向类型为 __fp16 的变量的指针可用作函数形参类型。
- __fp16 值可作为实际函数参数进行传递。这种情况下，必须转换为单精度值。
- __fp16 不能指定为函数的返回类型。但是，指向 __fp16 类型的指针可用作返回类型。

- 如果 __fp16 值用作返回 float 或 double 的函数的返回值，则会先转换为单精度或双精度值。

名称重整

C++ 对于半精度数据类型的名称重整是在 C++ 一般 ABI 中指定的。请参阅《ARM 体系结构的 C++ ABI》 (*C++ ABI for the ARM Architecture*)。

5.6.5 浮点计算和链接

了解浮点计算和浮点链接之间的区别是很重要的。浮点计算是由硬件协处理器指令或库函数执行的。浮点链接则与在使用浮点变量的函数之间传递参数的方式相关。

浮点链接的类型有：
- 软件浮点链接
- 硬件浮点链接。

软件浮点链接表示函数的参数和返回值通过 ARM 整数寄存器 r0 到 r3 以及堆栈进行传递。

硬件浮点链接使用 VFP 协处理器寄存器来传递参数和返回值。有关 VFP 协处理器寄存器的信息，请参阅 《ARM 体系结构的过程调用标准》 (*ARM Procedure Call Standard for the ARM Architecture*) (ARM IHI0042)。

使用软件浮点链接的好处在于，在有或没有 VFP 协处理器的核心中都可以运行所获得的代码。它不依赖于有没有 VFP 硬件协处理器，有没有 VFP 协处理器都可以使用它。

使用硬件浮点链接的好处在于，它比软件浮点链接更高效，但必须有 VFP 协处理器。

表 5-12 列出了所需浮点链接类型和浮点计算类型可用的编译器选项。

表 5-12 浮点链接和浮点运算的编译器选项

链接		计算		
硬件浮点链接	软件浮点链接	硬件浮点协处理器	软件浮点库 (fplib)	编译器选项
斲	是	斲	愉	`--fpu=softvfp`

斲	愉	愉	斲	`--fpu=softvfp+vfpv2`
				`--fpu=softvfp+vfpv3`
				`--fpu=softvfp+vfpv3_fp16`
				`--fpu=softvfp+vfpv3_d16`
				`--fpu=softvfp+vfp3_d16_fp16`

愉	斲	愉	斲	`--fpu=vfp`
				`--fpu=vfpv2`
				`--fpu=vfpv3`
				`--fpu=vfpv3_fp16`
				`--fpu=vfpv3_dp16`
				`--fpu=vfpv3_d16_fp16`

softvfp 指定软件浮点链接。如果使用软件浮点链接，则必须按照以下两种方式之一操作：

- 使用 --softvfp、 --fpu softvfp+vfpv2 或 --fpu softvfp+vfpv3、 --fpu softvfp+vfpv3_fp16、 softvfp+vfpv3_d16 或 softvfp+vfpv3_d16_fp16 选项之一编译调用函数和被调用函数
- 使用 __softfp 关键字声明调用函数和被调用函数。

--fpu softvfp、 --fpu softvfp+vfpv2、 --fpu softvfp+vfpv3、 --fpu softvfp+vfpv3_fp16、 --fpu softvfpv3_d16 和 --fpu softvfpv3_d16_fp16 选项均在整个文件中指定软件浮点链接。与之相对的是，利用 __softfp 关键字，可以对函数指定软件浮点链接。

注意

不是通过单独的编译器选项来分别选择所需的浮点链接类型和浮点计算类型，而是使用一个编译器选项 --fpu 同时选择两者。（请参阅表 5-12。）例如，--fpu=softvfp+vfpv2 选择软件浮点链接和硬件协处理器计算。每当使用 softvfp 时，都会指定软件浮点链接。

请参阅：

- 《编译器参考指南》中第2-61 页的*--fpu=name*
- 《编译器参考指南》中第4-15 页的*__softfp*
- 《编译器参考指南》中第4-67 页的*#pragma softfp_linkage*, *#pragma no_softfp_linkage*。

5.7 捕获和标识除零错误

消除代码中的除零错误非常重要，对于可能无法轻易恢复的嵌入式系统尤其如此。对于 ARM 处理器内核，除零错误属于以下类别：

- 整数除零错误
- （软件）浮点除零错误。

针对这两种情况，需要不同的技巧来捕获和标识这些错误。

5.7.1 整数除零

通过重新实现相应的 C 库辅助函数可以捕获和标识整数除零错误。

出现被零除时的缺省行为是，在使用信号函数或者重新实现 `__rt_raise` 或 `__aeabi_idiv0` 时调用 `__aeabi_idiv0`。否则除法函数会返回零。

`__aeabi_idiv0` 会发出带额外参数 `DIVBYZERO` 的 **SIGFPE**。

在代码中捕获除零错误

可使用以下方法捕获整数除零错误：

- 重新实现 C 库辅助函数 `__aeabi_idiv0`，使除零返回某种标准结果，例如零。

  通过 C 库辅助函数 `__aeabi_idiv` 和 `__aeabi_uidiv` 在代码中执行整数除法。两个函数均检查除零。

  检测到整数除零时，跳转到 `__aeabi_idiv0`。因此，要捕获除零，只需要在 `__aeabi_idiv0` 上放置一个断点。

  有关 AEABI 函数 `__aeabi_idiv`、`__aeabi_uidiv` 和 `__aeabi_idiv0` 的详细信息，请参阅 《ARM 体系结构的运行时 ABI》 (*Run-time ABI for the ARM Architecture*)。这可以在 `http://www.arm.com/products/DevTools/ABI.html` 上找到。

- 重新实现 C 库辅助函数 `__rt_raise` 来处理信号。

  缺省情况下，整数除零将发出信号。因此，要截取除零，您可以重新执行 `__rt_raise`。此函数的原型是：

  ```
  void __rt_raise(int signal, int type)
  ```

  发生除零错误时，`__aeabi_idiv0` 将调用 `__rt_raise(2, 2)`。因此，在 `__rt_raise` 的实现中，必须检查 `(signal == 2) && (type == 2)` 以确定是否发生了除零。

请参阅：

— 《库指南》中第2-24 页的*整数和浮点辅助函数*

— 《库指南》中第2-25 页的*使用 C 库*

— 《库指南》中第2-58 页的*调整错误信号、错误处理和程序退出*

— 《库指南》中第2-61 页的*__rt_raise()*。

- 使用信号函数安装 SIGFPE 的处理程序。这种方法的可移植性比前述其他方法更好，但效率更低。

在代码中标识除零错误

在进入 __aeabi_idiv0 时，链接寄存器 LR 包含了在应用程序代码中调用 __aeabi_uidiv 除法例程之后的指令的地址。要在源代码中标识出错行，仅需在调试器中查找 LR 提供的地址处的 C 代码行即可。

检查参数

如果要检查参数并将它们保存来供事后调试使用，可以捕获 __aeabi_idiv0。通过使用 $Super$$ 和 $Sub$$ 机制可以干预所有对 __aeabi_idiv0 的调用：

$Super$$ 给 __aeabi_idiv0 加上 $Super$$ 前缀，用于标识原始的未修补函数 __aeabi_idiv0。使用它可以直接调用原函数。

$Sub$$ 给 __aeabi_idiv0 加上 $Sub$$ 前缀，用于标识替代 __aeabi_idiv0 原版本要调用的新函数。使用它可以在原函数 __aeabi_idiv0 之前或之后添加处理。

示例 5-5 演示使用 $Super$$ 和 $Sub$$ 机制来截取 __aeabi_div0。请参阅 *《链接器用户指南》*中第4-14 页的*使用 $Super$$ 和 $Sub$$ 覆盖符号定义*。

示例 5-5 使用 $Super$$ 和 $Sub$$ 截取 __aeabi_div0

```
extern void $Super$$__aeabi_idiv0(void);
/* this function is called instead of the original __aeabi_idiv0() */
void $Sub$$__aeabi_idiv0()
{
    // insert code to process a divide by zero
    ...
    // call the original __aeabi_idiv0 function
    $Super$$__aeabi_idiv0();
}
```

5.7.2 （软件）浮点除法

通过组合使用内在函数和辅助函数，可以捕获和标识软件中的浮点除零错误。

在代码中捕获除零错误

要在代码中捕获浮点除零错误，请使用内在函数：

```
__ieee_status(FE_IEEE_MASK_ALL_EXCEPT, FE_IEEE_MASK_DIVBYZERO);
```

这会捕获代码中的所有除零错误，而不捕获所有其他异常，如示例 5-6 所示。

示例 5-6 除零错误

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

int main(void)
{
    float a, b, c;
    // Trap the Invalid Operation exception and untrap all other exceptions:
    __ieee_status(FE_IEEE_MASK_ALL_EXCEPT, FE_IEEE_MASK_DIVBYZERO);
    c = 0;
    a = b / c;
    printf("b / c = %f, ", a); // trap division-by-zero error
    return 0;
}
```

在代码中标识除零错误

任何时候出现异常均会调用 C 库辅助函数 _fp_trapveneer。在进入此函数时，寄存器的状态在异常发生之后便不更改。因此，要在包含导致异常的算法运算的应用程序代码中查找函数地址，仅需在函数 _fp_trapveneer 上放置断点并查看 LR 即可。

例如，假设示例 5-6 的 C 代码使用以下字符串从命令行编译：

```
armcc --fpmode ieee_full
```

反汇编编译器生成的汇编语言代码时， RealView Debugger 生成第5-41 页的示例 5-7 所示的输出。

示例 5-7 除零错误反汇编

```
main:
    00008080 E92D4010  PUSH     {r4,lr}
    00008084 E3A01C02  MOV      r1,#0x200
    00008088 E3A00C9F  MOV      r0,#0x9f00
    0000808C EB000F1A  BL       __ieee_status              <0xbcfc>
    00008090 E59F0020  LDR      r0,0x80b8
    00008094 E3A01000  MOV      r1,#0
    00008098 EB000DEA  BL       _fdiv                      <0xb848>
    0000809C EB000DBD  BL       _f2d                       <0xb798>
    000080A0 E1A02000  MOV      r2,r0
    000080A4 E1A03001  MOV      r3,r1
    000080A8 E28F000C  ADR      r0,{pc}+0x14 ; 0x80bc
    000080AC EB000006  BL       __0printf                  <0x80cc>
    000080B0 E3A00000  MOV      r0,#0
    000080B4 E8BD8010  POP      {r4,pc}
    000080B8 40A00000  <Data>   0x00 0x00 0xA0 '@'
    000080BC 202F2062  <Data>   'b' ' ' '/' ' '
    000080C0 203D2063  <Data>   'c' ' ' '=' ' '
    000080C4 202C6625  <Data>   '%' 'f' ',' ' '
    000080C8 00000000  <Data>   0x00 0x00 0x00 0x00
```

在 _fp_trapveneer 上放置断点并在调试监视器中执行反汇编将生成：

```
> go
Stopped at 0x0000BF6C due to SW Instruction BreakpointStopped at 0x0000BF6C:
TRAPV_S\_fp_trapveneer
```

然后，对寄存器的检查显示：

```
  r0: 0x40A00000     r1: 0x00000000     r2: 0x00000000     r3: 0x00000000
  r4: 0x0000C1DC     r5: 0x0000C1CC     r6: 0x00000000     r7: 0x00000000
  r8: 0x00000000     r9: 0x00000000    r10: 0x0000C0D4    r11: 0x00000000
 r12: 0x08000004     SP: 0x07FFFFF8     LR: 0x0000809C     PC: 0x0000BF6C
CPSR: nzcvIFtSVC
```

将链接寄存器 LR 中包含的地址设置为 0x809c，即在导致异常的指令 BL _fdiv 之后的指令的地址。

检查参数

要保存参数以供事后调试使用，您必须截取 _fp_trapveneer。要干预所有对 _fp_trapveneer 的调用，请使用 $Super$$ 和 $Sub$$ 机制。例如：

```
    AREA foo, CODE
IMPORT |$Super$$_fp_trapveneer|
EXPORT |$Sub$$_fp_trapveneer|
    |$Sub$$_fp_trapveneer|
;; Add code to save whatever registers you require here
;; Take care not to corrupt any needed registers
    B |$Super$$_fp_trapveneer|
    END
```

请参阅：

- 第5-38 页的*整数除零*
- *《链接器用户指南》*中第4-14 页的*使用$Super$$ 和$Sub$$ 覆盖符号定义*。

5.8 C99 的新功能

1999 C 标准将一系列新功能引入 C，包括：

- 新语言功能，包括新的关键字和标识符以及现有 C90 语言的扩展语法
- 新的库功能，包括新的库以及现有 C90 库的新宏和函数。

在以下部分中，我们将说明 C99 中初次使用的开发人员比较感兴趣的一系列新功能。

注意

C90 标准规定的语言是 C++ 的一个子集，从这个意义上来说，除了一些特殊情况外， C90 与标准 C++ 是兼容的。而 C99 标准中增加了一些新功能，从这个意义上来说， C99 不再与 C++ 兼容。

5.8.1 语言功能

C99 标准引入了几个新的语言功能，包括：

- 一些与 GNU 编译器中提供的 C90 扩展类似的功能，例如，带有可变数量的参数的宏。

  注意

  GNU 编译器中 C90 扩展的执行不一定与 C99 中类似功能的执行兼容。

- 一些 C++ 中的功能，例如， // 注释以及混合使用声明和代码的功能。
- 一些全新的功能，例如复数、受限指针和指定的初始值设定项。

下面几节将说明一些重要的 C99 新语言功能。

// 注释

可以使用 // 指示一行注释的开始，就像在 C++ 中一样。请参阅 *《编译器参考指南》*中第3-4 页的*// 注释*。

复合文字

ISO C99 支持复合文字。复合文字看上去象后面跟有初始值设定项的类型转换。其值是类型转换中指定的类型的对象，包含在初始值设定项中指定的元素。它是一个左值。例如：

```
int y[] = (int []) {1, 2, 3};
int z[] = (int [3]) {1};
```

指定的初始值设定项

在 C90 中，无法初始化数组、结构或联合的特定成员。C99 则支持通过使用*指定的初始值设定项*按名称或下标为数组、结构或联合的特定成员进行初始化。例如：

```
typedef struct
{
    char *name;
    int rank;
} data;
data vars[10] = { [0].name = "foo", [0].rank = 1,
                  [1].name = "bar", [1].rank = 2,
                  [2].name = "baz",
                  [3].name = "gazonk" };
```

在缺省情况下，未显式初始化的聚合成员将初始化为零。

十六进制浮点

C99 支持可以按十六进制格式写入的浮点数。例如：

```
float hex_floats(void)
{
    return 0x1.fp3; // 1 15/16 * 2^3
}
```

在十六进制格式下，指数是表示 2 次幂的十进制数，由此，有效部分将倍增。因此 `0x1.fp3 = 1.9375 * 8 = 1.55e1`。

可变数组成员

在具有多个成员的 **struct** 中， **struct** 的最后一个成员可以有不完整的数组类型。此类成员称为 **struct** 的*可变数组成员*。

——注意——

当 **struct** 包含可变数组成员时，整个 **struct** 本身将具有不完整的类型。

利用灵活的数组成员可以模拟 C 中的动态类型指定，可将数组大小的指定延迟到运行时处理。例如：

```
extern const int n;
typedef struct
{
    int len;
    char p[];
} str;
void foo(void){
    size_t str_size = sizeof(str);  // equivalent to offsetoff(str, p)
    str *s = malloc(str_size + (sizeof(char) * n));
}
```

__func__ 预定义标识符

__func__ 预定义标识符提供了一种获取当前函数名称的方法。例如，以下函数：

```
void foo(void)
{
    printf("This function is called '%s'.\n", __func__);
}
```

输出：

```
This function is called 'foo'.
```

inline 函数

C99 关键字 **inline** 提示编译器对使用 **inline** 限定的函数调用进行内联扩展。例如：

```
inline int max(int a, int b)
{
    return (a > b) ? a : b;
}
```

只有在可行的情况下，编译器才会内联 **inline** 限定的函数。如果内联函数对性能产生负面影响，则完全可以忽略提示。请参阅第5-16 页的*函数内联*。

——注意——

C99 中 **inline** 的语义与标准 C++ 中 **inline** 的语义不同。

long long 数据类型

C99 支持整型数据类型 **long long**。在 RVCT 中，此类型为 64 位宽。例如：

```
long long int j = 25902068371200;                // length of light day, meters
unsigned long long int i = 9460730472580800ULL; // length of light year, meters
```

请参阅 *《编译器参考指南》*中第3-7 页的*long long*。

带有可变数量的参数的宏

在 C99 中，可以声明接受可变数量的参数的宏。定义此类宏的语法类似于定义函数的语法。例如：

```
#define debug(format, ...) fprintf (stderr, format, __VA_ARGS__)
void Variadic_Macros_0()
{
    debug ("a test string is printed out along with %x %x %x\n", 12, 14, 20);
}
```

混合使用的声明和代码

C99 允许在复合语句中混合使用声明和代码，就像在 C++ 中一样。例如：

```
void foo(float i)
{
    i = (i > 0) ? -i : i;
    float j = sqrt(i);    // illegal in C90
}
```

选择和迭代语句的新的块范围

在 **for** 循环中，第一个表达式可以是声明，就像在 C++ 中一样。声明的范围仅扩展到循环体。例如：

```
extern int max;
for (int n = max - 1; n >= 0; n--)
{
    // body of loop
}
```

等效于：

```
extern int max;
{
    int n = max - 1;
    for (; n >= 0; n--)
```

```
    {
        // body of loop    }
}
```

注意

与 C++ 不同的是，不能在 **for** 测试、**if** 测试或 **switch** 表达式中引入新声明。

_Pragma 预处理运算符

C90 不允许作为宏扩展的结果而生成 #pragma 指令。利用 C99 _Pragma 运算符可以在 pragma 指令中嵌入预处理程序宏。例如：

```
# define RWDATA(X) PRAGMA(arm section rwdata=#X)
# define PRAGMA(X) _Pragma(#X)
RWDATA(foo)  // same as #pragma arm section rwdata="foo"
int y = 1;   // y is placed in section "foo"
```

限制的指针

使用 C99 关键字 **restrict** 可以确保不同的对象指针类型和函数参数数组不指向重叠的内存区域。这样，编译器可以执行优化，否则优化会因可能的混淆而被禁止。

在以下示例中，指针 a 与指针 b 指向的内存区域不同，也不能相同：

```
void copy_array(int n, int *restrict a, int *restrict b)
{
    while (n-- > 0)
        *a++ = *b++;
}
void test(void)
{
    extern int array[100];
    copy_array(50, array + 50, array);    // valid
    copy_array(50, array + 1, array);     // undefined behavior
}
```

不过，使用 **restrict** 限定的指针可以指向不同的数组，也可以指向同一数组中的不同区域。

5.8.2 库功能

C99 标准引入了一些程序员感兴趣的新的库功能，其中包括：

- 一些与 UNIX 标准库中提供的 C90 标准库扩展类似的功能，例如，snprintf 函数系列。
- 一些全新的库功能，例如，<fenv.h> 中提供的标准化浮点环境。

下面几节将说明一些重要的新的 C99 库功能。

<math.h> 中的附加数学库函数

C99 支持标准头文件 <math.h> 中的附加宏、类型和函数，而在相应的 C90 标准头文件中不存在这些宏、类型和函数。

C99 中提供而 C90 中不提供的新宏包括：

```
INFINITY // positive infinity
NAN      // IEEE not-a-number
```

C99 中提供而 C90 中不提供的新的通用函数宏包括：

```
#define isinf(x) // non-zero only if x is positive or negative infinity
#define isnan(x) // non-zero only if x is NaN
#define isless(x, y) // 1 only if x < y and x and y are not NaN, and 0 otherwise
#define isunordered(x, y) // 1 only if either x or y is NaN, and 0 otherwise
```

C99 中提供而 C90 中不提供的新的数学函数包括：

```
double acosh(double x); // hyperbolic arccosine of x
double asinh(double x); // hyperbolic arcsine of x
double atanh(double x); // hyperbolic arctangent of x
double erf(double x); // returns the error function of x
double round(double x); // returns x rounded to the nearest integer
double tgamma(double x); // returns the gamma function of x
```

C99 支持所有实数浮点类型的新的数学函数。

此外，还支持全部现有 <math.h> 函数的单精度版本。

复数

在 C99 模式中，编译器支持复数和虚数。在 GNU 模式中，编译器只支持复数。

例如：

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

int main(void)
{
    complex float z = 64.0 + 64.0*I;
    printf( "z = %f + %fI\n" , creal(z), cimag(z));
    return 0;
}
```

复数类型有：

- **float complex**
- **double complex**
- **long double complex**。

布尔类型和 <stdbool.h>

C99 引入了本机类型 **_Bool**。对于布尔测试，相关的标准头文件 <stdbool.h> 引入了宏 bool、 true 和 false。例如：

```
#include <stdbool.h>
bool foo(FILE *str)
{
    bool err = false;
    ...
    if (!fflush(str))
    {
        err = true;
    }
    ...
    return err;
}
```

注意

C99 的布尔语义旨在与 C++ 的语义相匹配。

<inttypes.h> 和 <stdint.h> 中的扩展整数类型和函数

在 C90 中，**long** 数据类型可以作为最大的整数类型，也可以作为 32 位容器。而在 C99 中，通过新的标准库头文件 <inttypes.h> 和 <stdint.h> 避免了这种歧义。

头文件 <stdint.h> 引入了以下新类型：

- intmax_t 和 uintmax_t，这些是最大宽度的有符号和无符号整数类型
- intptr_t 和 unintptr_t，这些是可以保留有符号和无符号对象指针的整数类型。

头文件 <inttypes.h> 提供用于操作 intmax_t 类型值的库函数，其中包括：

```
intmax_t imaxabs(intmax_t x); // absolute value of x
imaxdiv_t imaxdiv(intmax_t x, intmax_t y) // returns the quotient and remainder
                                          // of x / y
```

<fenv.h> 中的浮点环境访问

通过 C99 标准头文件 <fenv.h> 可以访问用于数字编程的 IEEE 754 兼容浮点环境。该库引入了两种类型和大量的宏用于管理和控制浮点状态。

支持的新类型包括：

- fenv_t，表示整个浮点环境
- fexcept_t，表示浮点状态。

支持的新宏包括：

- FE_DIVBYZERO、 FE_INEXACT、 FE_INVALID、 FE_OVERFLOW 和 FE_UNDERFLOW，用于管理浮点异常
- FE_DOWNWARD, FE_TONEAREST, FE_TOWARDZERO, FE_UPWARD，用于管理代表的舍入方向上的舍入
- FE_DFL_ENV，表示缺省浮点环境。

新函数包括：

```
int feclearexcept(int ex); // clear floating-point exceptions selected by ex
int feraiseexcept(int ex); // raise floating point exceptions selected by ex
int fetestexcept(int ex); // test floating point exceptions selected by x
int fegetround(void); // return the current rounding mode
int fesetround(int mode); // set the current rounding mode given by mode
int fegetenv(fenv_t *penv); return the floating-point environment in penv
int fesetenv(const fenv_t *penv); // set the floating-point environment to penv
```

<stdio.h> 中的 snprintf 函数系列

使用 C90 标准头文件 <stdio.h> 中的 sprintf 函数系列可能会有危险。在以下语句中：

```
sprintf(buffer, size, "Error %d: Cannot open file '%s'", errno, filename);
```

变量 size 指定要在 buffer 中插入的最小字符数。因此，输出的字符数可能会大于分配给字符串的内存中可容纳的字符数。

在 C99 版本的 <stdio.h> 中找到的 snprintf 函数是安全版本的 sprintf 函数，可以防止缓冲区溢出。在以下语句中：

```
snprintf(buffer, size, "Error %d: Cannot open file '%s'", errno, filename);
```

变量 size 指定要在 buffer 中插入的最大字符数。在缓冲区大小总是大于 size 指定的大小的情况下，缓冲区永远不会溢出。

<tgmath.h> 中的泛型数学宏

新的标准头文件 <tgmath.h> 定义了几个数学函数系列，它们属于泛型类型，也就是按浮点类型进行重载。例如，三角函数 cos 的工作方式就如同包含以下重载声明一样：

```
extern float cos(float x);
extern double cos(double x);
extern long double cos(long double x);
...
```

以下语句：

```
p = cos(0.78539f); // p = cos(pi / 4)
```

调用的是单精度 cos 函数，这是由文本 0.78539f 的类型确定的。

——注意——

在 C++ 中，可以使用运算符重载机制定义泛型类型的数学函数系列。使用 C++ 中运算符重载定义的泛型类型函数系列的语义不同于在 <tgmath.h> 中定义的相应泛型类型函数系列的语义。

<wchar.h> 中的宽字符 I/O 函数

C99 中引入了宽字符 I/O 函数。这些函数用于在文件中读取和写入宽字符，读取和写入的方式与普通字符一样。ARM C 库支持 wchar.h 中定义的所有 C99 函数。请参阅 *ISO/IEC 9899:TC2* 的第 7.24 节。

第 6 章
诊断消息

ARM 编译器发出有关潜在的可移植性问题和其他危险的消息。利用本节介绍的编译器选项，您可以：

- 关闭特定消息。例如，在移植以老式 C 书写的程序的早期，可关闭警告。通常，与关闭消息相比，检查代码更好一些。
- 更改特定消息的严重性。

本节包括以下小节：

6.1 重定向诊断

使用 `--errors=filename` 选项将编译器诊断输出重定向到一个文件中。不重定向与命令选项相关的诊断。

请参阅第6-4 页的*控制诊断消息的输出*。

6.2 诊断消息的严重性

如表 6-1 所述，诊断消息具有相关联的*严重性*。

表 6-1 诊断消息的严重性

严重性	说明
内部故障	内部故障指示编译器的内部问题。有关第 xiii 页上的 RealView 编译工具的反馈中列出的信息，请与供应商联系。
错误	错误指示导致编译终止的问题。这些错误包括命令行错误、内部错误、丢失包含文件和违反 C 或 C++ 语言的语法或语义规则。如果指定了多个源文件，则不编译更多的源文件。
警告	警告指示代码中存在异常情况，可能有问题。编译将会继续，除非检测到严重性为“错误”级别的其他问题，否则将生成对象代码。
备注	备注指示 C 或 C++ 的常规但有时是非习惯性的用法。缺省情况下不显示这些诊断消息。编译将会继续，除非检测到严重性为 “错误”级别的其他问题，否则将生成对象代码。

6.3 控制诊断消息的输出

使用这些选项可以控制诊断消息的输出：

`--no_brief_diagnostics`, `--brief_diagnostics`

启用或禁用使用短格式显示诊断输出的模式。启用时不显示原始源语句行，并且当错误消息文本太长、一行放不下时也不换行。缺省为 `--no_brief_diagnostics`。

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

指定用于显示诊断消息的样式。

`--errors=`*`filename`*

将诊断消息输出从 `stderr` 重定向到指定的错误文件 *`filename`*。当系统不能很好地支持文件的输出重新定向时，该选项很有用。

`--remarks` 导致编译器发出备注消息，例如，结构中的填充警告。缺省情况下不发出备注消息。

`--no_wrap_diagnostics`, `--wrap_diagnostics`

在错误消息文本太长，一行放不下时，启用或禁用错误消息文本的换行。

请参阅 《编译器参考指南》中第2-2 页的*命令行选项*。

6.4 更改诊断消息的严重性

利用这些选项，可更改所有备注和警告以及数量有限的错误的诊断严重性：

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

将具有指定标签的诊断消息的严重性设置为 “错误”。

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

将具有指定标签的诊断消息的严重性设置为 “备注”。

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

将具有指定标签的诊断消息的严重性设置为 “警告”。

这些选项要求提供要更改的错误消息的逗号分隔列表。例如，由于缺省情况下不显示备注，您可能希望将编号为 #1293 的警告消息的严重性更改为 “备注”。

要执行此操作，请使用以下命令：

```
armcc --diag_remark=1293 ...
```

——注意——

这些选项也具有对应的等效编译指示。请参阅第4-13 页的*编译指示*。

可以更改以下诊断消息：

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

6.5 禁止显示诊断消息

若要禁止显示所有具有指定标签的诊断消息，请使用下面的选项：
--diag_suppress=*tag*[, *tag*, ...]

另请参阅：

- 第4-13 页的*编译指示*
- *《编译器参考指南》*中第2-46 页的*--diag_suppress=tag[,tag,...]*。

6.6 诊断消息中的前缀字母

RVCT 工具自动将标识字母插入诊断消息，如表 6-2 中所述。借助这些前缀字母， RVCT 工具可使用重叠的消息范围。

表 6-2 标识诊断消息

前缀字母	RVCT 工具
C	armcc
A	armasm
L	armlink 或 armar
Q	fromelf

以下规则适用：

- 无前缀的消息编号可由所有 RVCT 工具处理。
- 有前缀的消息编号仅由具有匹配前缀的工具处理。
- 工具不对前缀不匹配的消息进行处理。

这样，编译器前缀 C 可以与 --diag_error、--diag_remark 和 --diag_warning 一起使用或在禁用显示消息时使用，例如：

```
armcc --diag_suppress=C1287,C3017 ...
```

使用前缀字母可以控制从编译器传递到其他工具的选项，例如，包含前缀字母 L 可指定链接器消息编号。

6.7 使用 -W 禁止显示警告消息

-W 选项可禁止显示所有警告。

6.8 退出状态代码和终止消息

如果编译器在编译期间检测到任何警告或错误，则编译器会将消息写入 stderr。在消息结尾处将显示汇总消息，它提供每种类型消息的总数，其格式为：

filename: *n* warnings, *n* errors

其中 *n* 表示检测到的警告或错误的数量。

注意

缺省情况下不显示备注。要显示备注，可使用 --remarks 编译器选项。如果只生成备注消息，则不显示汇总消息。

本节还包括以下内容：

- *信号反应*
- *退出状态*

6.8.1 信号反应

编译器捕获信号 **SIGINT**（由用户中断引起，如 ^C）和 **SIGTERM**（由 UNIX kill 命令引起），并且导致异常终止。

6.8.2 退出状态

完成时，如果检测到错误，则编译器返回一个大于 0 的值。如果未检测到任何错误，则返回值为 0。

有关编译器如何处理不同级别的诊断消息的详细信息，请参阅第6-3 页的*诊断消息的严重性*。

6.9 数据流警告

编译器将数据流分析作为优化进程的一部分执行。此信息可用于确定代码中的潜在问题，例如，发出有关使用未初始化变量的警告。

数据流分析只能发出有关保存在处理器寄存器中的局部变量的警告，不能发出有关保存在内存中的全局变量或存放在堆栈中的变量或结构的警告。

您必须了解：

- 缺省情况下将发出数据流警告（在 RVCT v2.0 及更早版本中，只有在指定 `-fa` 选项后才会发出数据流警告）。
- 在 `-O0` 级别下禁用数据流分析 （即使指定了 `-fa` 选项）。

此分析的结果将随使用的优化级别而有所不同。这意味着，较高优化级别可能生成很多在较低优先级不会出现的警告。例如，下面的源代码会导致编译器在 `-O2` 级别生成警告 `C3017W: i may be used before being set`：

```
int f(void)
{
    int i;
    return i++;
}
```

数据流分析不能可靠地确定错误代码，编译器发出的所有 `C3017W` 警告都只是为了指出可能存在的问题。若要对代码进行完全分析，请使用 `--diag_suppress=C3017` 禁止显示此警告，然后使用任何适当的第三方分析工具（如 Lint）。

第 7 章
使用内联汇编器和嵌入式汇编器

本章介绍了 ARM 编译器 -armcc 的优化内联汇编器和非优化嵌入式汇编器。本章分为以下几节：

7.1 内联汇编器

ARM 编译器提供了内联汇编器，利用内联汇编器可以编写优化的汇编语言例程，并可使用不能从 C 或 C++ 获得的目标处理器功能。

本节分为以下几个小节：

- *内联汇编器支持*
- 第7-3 页的*内联汇编器语法*
- 第7-5 页的*内联汇编操作的限制*
- 第7-8 页的*虚拟寄存器*
- 第7-8 页的*常数*
- 第7-9 页的*指令扩展*
- 第7-10 页的*条件标记*
- 第7-10 页的*操作数*
- 第7-12 页的*函数调用和跳转*
- 第7-13 页的*标签*
- 第7-14 页的*与先前版本 ARM C/C++ 编译器的差异*

另请参阅：

- *《开发指南》*中的第 4 章 *混合使用 C、C++ 和汇编语言*，了解如何在 C、C++ 源代码中使用内联汇编器的信息以及内联汇编语言的限制
- *《汇编器指南》*，以详细了解如何为 ARM 处理器编写汇编程序。

7.1.1 内联汇编器支持

内联汇编器只支持 ARM 汇编语言，而不支持以下语言和指令：

- Thumb 汇编语言
- Thumb-2 汇编语言
- ARMv7 指令
- VFP 指令
- NEON 指令。

您可使用嵌入式汇编器来支持 Thumb 和 Thumb-2。

内联汇编器支持大多数 ARMv6 指令，包括完整的 ARMv6 SIMD 指令集。内联汇编器不支持的 ARMv6 指令为 `SETEND` 和一些系统扩展。

内联汇编器支持大部分 ARMv5 指令，包括通用协处理器指令。内联汇编器不支持的 ARMv5 指令为 BX、BLX 和 BXJ。

7.1.2 内联汇编器语法

ARM 编译器支持 **asm** 关键字 (C++) 或 **__asm** 关键字 （C 和 C++）引入的一种扩展内联汇编器语法。以下各节介绍了这些关键字的语法：

- *含有 __asm 关键字的内联汇编*
- *含有 asm 关键字的内联汇编*
- 第7-4 页的*使用 __asm 和 asm 的规则*

在任何语句位置，都可以使用 **asm** 或 **__asm** 语句。

含有 __asm 关键字的内联汇编

内联汇编器使用汇编器说明符进行调用，后面跟用大括号或括号括起来的汇编器指令列表。您可以指定使用以下格式的内联汇编器代码：

- 在单行中，示例如下：

  ```
  __asm("instruction[;instruction]"); // Must be a single string
  __asm{instruction[;instruction]}
  ```

  不能包括注释。

- 在多行中，示例如下：

  ```
  __asm
  {
      ...
      instruction
      ...
  }
  ```

  在内联汇编语言块中的任何位置，都可以使用 C 或 C++ 注释。

另请参阅第7-4 页的*使用 __asm 和 asm 的规则*。

含有 asm 关键字的内联汇编

编译 C++ 时，ARM 编译器支持 ISO C++ 标准中建议的 **asm** 语法。您可以指定使用以下格式的内联汇编器代码：

- 在单行中，示例如下：

  ```
  asm("instruction[;instruction]"); // Must be a single string
  asm{instruction[;instruction]}
  ```

不能包括注释。

- 在多行中，示例如下：

```
asm
{
    ...
    instruction
    ...
}
```

在内联汇编语言块中的任何位置，都可以使用 C 或 C++ 注释。

使用 __asm 和 asm 的规则

使用 **__asm** 和 **asm** 关键字时，请遵循以下规则：

- 如果在同一行中包括多个指令，则必须用分号 (;) 进行分隔。如果使用双引号，则必须将所有指令包含在一对双引号 (") 内。
- 如果一条指令需要占用多行，则必须用反斜杠符号 (\) 指定后续行。
- 对于多行格式，可以在内联汇编语言块中的任何位置使用 C 或 C++ 注释。但当一行中包含多个指令时，不能嵌入注释。
- 在汇编语言中，逗号 (,) 用作分隔符，因此使用逗号运算符的 C 表达式必须括在括号中，以区分二者：

```
__asm
{
    ADD x, y, (f(), z)
}
```

- **asm** 语句必须位于 C++ 函数内。可在任何应该出现 C++ 语句的地方使用 **asm** 语句。
- 在内联汇编器中，寄存器名视为 C 或 C++ 变量。它们不一定与相同名称的物理寄存器相关 （请参阅第7-8 页的*虚拟寄存器*）。如果未将寄存器声明为 C 或 C++ 变量，编译器将生成一个警告。
- 在内联汇编器中不要保存和恢复寄存器。编译器会为您完成此操作。此外，内联汇编器不提供对物理寄存器的直接访问。请参阅第7-8 页的*虚拟寄存器*。

  如果未向寄存器写入内容就读取寄存器 （CPSR 和 SPSR 除外），则会发出一条错误消息。例如：

```
int f(int x)
{
    __asm
    {
        STMFD sp!, {r0}    // save r0 - illegal: read before write
        ADD r0, x, 1
        EOR x, r0, x
        LDMFD sp!, {r0}    // restore r0 - not needed.
    }
    return x;
}
```

该函数必须按以下形式编写：

```
int f(int x)
{
    int r0;
    __asm
    {
        ADD r0, x, 1
        EOR x, r0, x
    }
    return x;
}
```

请参阅*内联汇编操作的限制*。

7.1.3 内联汇编操作的限制

可在内联汇编代码中执行的操作存在许多限制。这些限制提供了安全的方法，并确保在汇编代码中不违反已编译的 C 和 C++ 代码中的假设。

其他限制

内联汇编器具有以下限制：

- 内联汇编器是一种高级汇编器，它生成的代码可能不总是与您编写的代码完全一致。不能用它来生成比编译器生成的代码更有效的代码。应当使用嵌入式汇编器或 ARM 汇编器 `armasm` 来实现此目的。
- 不支持 ARM 汇编器 `armasm` 中提供的某些低级功能，如跳转和写入 PC。
- 不支持标签表达式。
- 不能使用点表示法 (.)或 `{PC}` 获取当前指令的地址。
- 不能使用 `&` 运算符表示十六进制常数。应改为使用 `0x` 前缀。例如：

  ```
  __asm { AND x, y, 0xF00 }
  ```

- 用于指定 8 位常数实际循环的表示法在内联汇编语言中不能使用。这意味着在使用 8 位移位常数时，如果更新了 NZCV 标记，C 标记必须视为已破坏。
- 不得修改堆栈。这是没有必要的，因为编译器会根据需要自动堆叠和恢复任何工作寄存器。编译器不允许显式堆叠和恢复工作寄存器。

寄存器

必须小心使用寄存器（如 r0-r3、sp 和 lr）和 CPSR 中的 NZCV 标记。如果使用 C 或 C++ 表达式，这些寄存器可能被用作临时寄存器，并且 NZCV 标记可能在计算表达式时被编译器破坏。请参阅第7-8 页的*虚拟寄存器*。

因为不能直接访问任何物理寄存器，所以无法用内联汇编代码显式读取或修改 pc、lr 和 sp。不过，您可以使用 *《编译器参考指南》*中介绍的以下内在函数访问这些寄存器：

- 第4-74 页的*__current_pc*
- 第4-74 页的*__current_sp*
- 第4-89 页的*__return_address*

处理器模式

可以更改处理器模式或修改协处理器状态，但编译器无法识别这些更改。如果更改处理器模式，则直到改回到原模式后才能使用 C 或 C++ 表达式，否则编译器将破坏新处理器模式的寄存器。

同样，如果通过执行浮点指令更改了浮点协处理器状态，则直到恢复原状态后才能使用浮点表达式。

Thumb 指令集

为 Thumb 状态编译 C 或 C++ 时，内联汇编器不可用且不汇编 Thumb 指令。相反，编译器会自动切换至 ARM 状态。

如果要在包含为 Thumb 编译的代码的源文件中包括内联汇编，则在 #pragma arm 和 #pragma thumb 语句之间包括含有内联汇编器代码的函数。例如：

```
...         // Thumb code
#pragma arm // ARM code. Switch code generation to the ARM instruction set so
            // that the inline assembler is available.

int add(int i, int j)
{
    int res;
```

```
    __asm
    {
        ADD   res, i, j   // add here
    }
    return res;
}
#pragma thumb   // Thumb code. Switch back to the Thumb instruction set.
                // The inline assembler is no longer available.
```

还必须使用 --apcs /interwork 编译器选项编译代码。

请参阅：

- 第2-24 页的*交互操作限定符*
- *《编译器参考指南》*中第4-55 页的*编译指示*。

VFP 协处理器

内联汇编器不提供对 VFP 指令的直接支持。不过，可以使用通用协处理器指令指定这些指令。

不得使用内联汇编代码更改 VFP 向量模式。内联汇编可能包含可使用编译器生成的 VFP 代码进行计算的浮点表达式操作数。因此，仅编译器可修改 VFP 状态是非常重要的。

不支持的指令

内联汇编器不支持以下指令：

- BKPT、 BX、 BXJ 和 BLX 指令

  注意

  可以使用 __breakpoint() 内在函数，在 C 和 C++ 代码中插入 BKPT 指令。

- LDR Rn, =*expression* 伪指令。应改用 MOV Rn, *expression*。（这可以生成从文字池进行的加载操作。）
- LDRT、 LDRBT、 STRT 和 STRBT 指令
- MUL、 MLA、 UMULL、 UMLAL、 SMULL 和 SMLAL 标记设置指令
- MOV 或 MVN 标记设置指令，其中第二个操作数为常数
- 用户模式 LDM 指令
- ADR 和 ADRL 伪指令。

请参阅 《编译器参考指南》中第4-71 页的__breakpoint

7.1.4 虚拟寄存器

内联汇编器不提供对 ARM 处理器物理寄存器的直接访问。如果在内联汇编器中将 ARM 寄存器名称用作操作数，它就成为对具有相同名称的虚拟寄存器的引用，而不是对物理 ARM 寄存器的引用。

在优化和代码生成过程中，编译器给每个虚拟寄存器分配相应的物理寄存器。不过，汇编代码中使用的物理寄存器可能与在指令中指定的不同。可将这些虚拟寄存器显式定义为标准 C 或 C++ 变量。如果这些虚拟寄存器未定义，编译器会为它们提供隐式定义。

编译器定义的虚拟寄存器具有函数局部范围，即在单个函数内，引用同一虚拟寄存器名称的多个 **asm** 语句或声明访问的是同一个虚拟寄存器。

没有为 sp (r13)、lr (r14) 和 pc (r15) 寄存器创建虚拟寄存器，而且不能在内联汇编代码中读取或直接修改它们。有关如何修改源代码的信息，请参阅第7-25 页的 *访问 sp、 lr 或 pc 的旧内联汇编器*。

不存在虚拟 *处理器状态寄存器* (PSR)。任何对 PSR 的引用总是指向物理 PSR。

符合以前所述准则的现有内联汇编器代码继续执行与编译器先前版本相同的功能，但在每个指令中使用的实际寄存器可能不同。

每个虚拟寄存器中的初值是不可预测的。必须在读之前向虚拟寄存器写入初值。如果在写入之前试图读取虚拟寄存器 （例如试图读取与变量 r1 相关联的虚拟寄存器），编译器将会生成错误。

您还必须在 C 或 C++ 代码中明确声明变量的名称。最好将 C 或 C++ 变量用作指令操作数。在第一次使用虚拟或物理寄存器名称时编译器会生成一个警告，每个转换单元仅生成一次。例如，如果指定寄存器 r3，则将显示一个警告。

7.1.5 常数

常数表达式说明符 # 是可选的。如果使用了它，其后的表达式必须是常数。

7.1.6 指令扩展

内联汇编代码中的 ARM 指令可能会在编译对象中扩展为几条指令。扩展取决于指令、指令中指定的操作数个数以及每个操作数的类型和值。

使用常数的指令

带有常数操作数的指令中的常数不仅仅局限于该指令所允许的值。相反，编译器会将指令转换为具有相同效果的指令序列。例如：

```
ADD r0,r0,#1023
```

可能解释为：

```
ADD r0,r0,#1024
SUB r0,r0,#1
```

除了协处理器指令之外，所有带有常数操作数的 ARM 指令都支持指令扩展。此外，当 MUL 指令的第三个操作数是常数时，该指令可以扩展为一系列加法和移位指令。

使用扩展指令更新 CPSR 的效果是：

- 算法指令正确设置 NZCV 标记
- 逻辑指令：
  - 正确设置 NZ 标记
  - 不改变 V 标记
  - 破坏 C 标记。

加载和存储指令

LDM、STM、LDRD 和 STRD 指令可替换为等效的 ARM 指令。在这种情况下，编译器输出一条警告消息，通知您它可能扩展指令。

编写内联汇编代码的方式必须是不依赖于期望的指令条数或每条指定指令期望的执行时间。

通常约束操作数寄存器对的指令，如 LDRD 和 STRD，被具有等效功能而并无约束的指令序列替换。不过，这些指令可重新组合成 LDRD 和 STRD 指令。

所有的 LDM 和 STM 指令被扩展为等效的 LDR 和 STR 指令序列。不过，在优化过程中，编译器可能因此将单独的指令重新组合为一条 LDM 或 STM 指令。

7.1.7 条件标记

内联汇编指令可能显式或隐式地试图更新处理器条件标记。有些内联汇编指令仅包含虚拟寄存器操作数或简单表达式操作数（请参阅*操作数*），其行为可以预见。如果指定了隐式或显式更新，则由指令设置条件标记。如果未指定更新，则条件标记不会更改。指令操作数中的任何一个都不是简单操作数时，除非指令更新条件标记，否则条件标记可能会被破坏。一般情况下，编译器不易诊断出对条件标记的潜在破坏。不过，对于需要构造随后析构 C++ 临时函数的操作数，如果指令试图更新条件标记，编译器将显示警告。这是因为析构可能会破坏条件标记。

7.1.8 操作数

操作数可以是多种类型之一：

虚拟寄存器

在内联汇编指令中指定的寄存器总表示虚拟寄存器而不是物理 ARM 整数寄存器。虚拟寄存器不需要声明，其大小与物理寄存器相同。不过，汇编代码中使用的物理寄存器可能与在指令中指定的不同。请参阅第7-8 页的*虚拟寄存器*。

表达式操作数

在内联汇编指令中，可将函数参数、C 或 C++ 变量和其他 C 或 C++ 表达式指定为寄存器操作数。

代替 ARM 整数寄存器的表达式类型必须为除 `long long` 之外的整数类型（即 `char`、`short`、`int` 或 `long`）或指针类型。对 `char` 或 `short` 类型不执行符号扩展。您必须对这些类型执行显式符号扩展。编译器可能会添加代码，以计算这些表达式并将它们分配给寄存器。

当操作数用作目标时，如果在修改寄存器的位置将表达式用作操作数，则表达式必须为可修改的左值。例如，基址寄存器的目标寄存器或基址寄存器发生更新。

对于包含多个表达式操作数的指令，没有指定表达式操作数求值的顺序。

只有满足了指令的条件，才能对条件指令的表达式操作数求值。

将 C 或 C++ 表达式用作内联汇编器操作数，可能会导致一条指令被扩展为多条指令。如果表达式的值不能满足 *《ARM 体系结构参考手册》*中阐明的指令操作数约束，就会发生这种情况。

如果用作操作数的表达式创建需要析构的临时函数，则析构发生在执行内联汇编指令之后。这与 C++ 析构临时函数的规则相类似。

简单表达式操作数是以下类型之一：

- 变量值
- 变量地址
- 指针变量的反引用
- 编译时常数。

包含以下类型之一的任何表达式都不是简单表达式操作数：

- 隐式函数调用，如除法，或显式函数调用
- C++ 临时函数的构造
- 算术或逻辑运算。

寄存器列表

寄存器列表最多可包含 16 个操作数。这些操作数可以是虚拟寄存器或表达式寄存器操作数。

在寄存器列表中指定虚拟寄存器和表达式操作数的顺序很重要。寄存器列表中操作数的读写顺序是从左到右。第一个操作数使用最低地址，随后的操作数的地址依次在前一地址基础上增加 4。在 LDM 或 STM 指令的普通操作中，编号最低的物理寄存器总是存入最低的内存地址，此新行为与之相反。这个行为上的区别是寄存器虚拟化的结果。

表达式操作数或虚拟寄存器可以在寄存器列表中出现多次，并且在每次指定后使用。

如果指定基址寄存器，则进行更新。在内存加载操作过程中，更新操作将改写加载到基址寄存器的所有值。

在特权模式下，内联汇编器不支持通过在寄存器列表后指定 ^ 来对用户模式下的寄存器进行操作。

中间操作数

在内联汇编指令中，可能将 C 或 C++ 整型常数表达式用作立即值。

用于指定立即数移位的常数表达式的值必须位于 *《ARM 体系结构参考手册》*中定义的适用于移位操作的范围内。

用于为内存或协处理器数据传送指令指定直接偏移量的常数表达式，必须有一个适当对齐的值。

7.1.9 函数调用和跳转

利用内联汇编器的 BL 和 SVC 指令，可在常规指令字段后指定 3 个可选列表。这些指令的格式如下：

```
SVC{cond} svc_num, {input_param_list}, {output_value_list}, {corrupt_reg_list}
BL{cond} function, {input_param_list}, {output_value_list}, {corrupt_reg_list}
```

注意

SVC 指令以前的名称是 SWI。内联汇编器仍然接受用 SWI 代替 SVC。

以下各节介绍这些列表：

- *未指定任何列表*
- *输入参数列表*
- 第7-13 页的*输出值列表*
- 第7-13 页的*被破坏的寄存器的列表*

注意

- 内联汇编器不支持 BX、BLX 和 BXJ 指令。
- 不能在任何输入、输出或被破坏的寄存器列表中指定 lr、sp 或 pc 寄存器。
- 任何 SVC 指令或函数调用不能更改 sp 寄存器。

未指定任何列表

如果未指定任何列表，则：

- 将 r0-r3 用作输入参数
- 将 r0 用于输出值
- r12 和 r14 可能损坏。

输入参数列表

此列表指定作为函数调用或 SVC 指令的输入参数的表达式或变量，以及包含表达式或变量的物理寄存器。它们被指定为对物理寄存器的赋值或物理寄存器名称。单一列表中可包含输入寄存器规范的两种类型。

内联汇编器确保在输入 BL 或 SVC 指令之前，指定的物理寄存器中存在正确的值。指定物理寄存器名称而并不赋值，这样确保物理寄存器中可存在相同名称的虚拟寄存器中的值。这确保了与现有内联汇编器代码的向后兼容性。

例如，指令：

```
BL foo, { r0=expression1, r1=expression2, r2 }
```

生成以下伪代码：

```
MOV (physical) r0, expression1
MOV (physical) r1, expression2
MOV (physical) r2, (virtual) r2
BL foo
```

输出值列表

此列表指定的物理寄存器包含来自 BL 或 SVC 的输出值以及这些输出值必须存储到的目标位置。输出值被指定为从物理寄存器到可修改左值表达式的赋值，或被指定为单个物理寄存器名称。

内联汇编器从指定的物理寄存器中取值并赋值到指定的表达式中。指定物理寄存器名称而并不赋值，会导致相同名称的虚拟寄存器被物理寄存器中的值更新。

例如，指令：

```
BL foo, { }, { result1=r0, r1 }
```

生成以下伪代码：

```
BL foo
MOV result1, (physical) r0
MOV (virtual) r1, (physical) r1
```

被破坏的寄存器的列表

此列表指定被所调用函数破坏的物理寄存器。如果条件标记被调用的函数修改，则必须在被破坏的寄存器的列表中指定 PSR。

BL 和 SVC 指令总是破坏 lr。

如果此列表被忽略，则对于 BL 和 SVC，寄存器 r0-r3、 pc、 lr 和 PSR 被破坏。

跳转指令 B 只能用于跳转到单个 C 或 C++ 函数内的标签。

7.1.10 标签

内联汇编代码中定义的标签可用作跳转或 C 和 C++ goto 语句的目标。在内联汇编代码中， C 和 C++ 中定义的标签可通过以下形式用作跳转指令的目标：

```
BL{cond} label
```

7.1.11 与先前版本 ARM C/C++ 编译器的差异

ARM 编译器与先前版本 ARM C 和 C++ 编译器中的内联汇编器有显著差异。本节着重介绍两者的主要差异。有关将现有汇编代码用于内联汇编器的详细信息，请参阅 *《RealView 编译工具开发指南》*。

ARMv6 指令

在所有 ARMv6 指令中，内联汇编器只支持 ARMv6 媒体指令。

虚拟寄存器

编译器的内联汇编代码总是指定虚拟寄存器。编译器选择在代码生成过程中用于每条指令的物理寄存器，并让编译器完全优化汇编代码和周围的 C 或 C++ 代码。

`pc` (`r15`)、`lr` (`r14`) 和 `sp` (`r13`) 寄存器根本不能访问。访问这些寄存器时，会产生错误消息。

虚拟寄存器的初值未定义。因此，必须在读之前先写入虚拟寄存器。如果代码在写入之前读取虚拟寄存器，编译器将发出警告。如果内联汇编代码起始处出现依赖于物理寄存器中特定值的旧代码，编译器也会生成这些警告，例如：

```
int add(int i, int j)
{
    int res;
    __asm
    {
        ADD res, r0, r1   // relies on i passed in r0 and j passed in r1
    }
    return res;
}
```

此代码生成警告和错误消息。

这是由于它在写入虚拟寄存器 `r0` 和 `r1` 之前读取它们，因而生成了错误消息。生成警告消息是因为 `r0` 和 `r1` 必须定义为 C 或 C++ 变量。改正的代码是：

```
int add(int i, int j)
{
    int res;
    __asm
    {
        ADD res, i, j
    }
    return res;
}
```

指令扩展

编译器中的内联汇编器将指令 LDM、STM、LDRD 和 STRD 扩展为执行等效功能的单寄存器内存操作的序列。

编译器可能优化单寄存器内存操作指令序列，回到多寄存器内存操作。

寄存器列表

LDM 或 STM 指令的寄存器列表中操作数的顺序很重要。它们按给定的顺序依次使用，即从左到右，并且第一个操作数引用生成的最低内存地址。这与先前编译器的行为相反，在先前编译器行为中，编号最低的寄存器总是引用由指令生成的最低内存地址。

因为现在可以在寄存器列表中与虚拟寄存器一起使用表达式操作数，所以会发生这种变化。如果编译器遇到仅包含虚拟寄存器的寄存器列表，而在此列表中新的排序结果与先前版本的 ARM C 和 C++ 编译器有所不同，则会显示警告消息。

Thumb 指令

编译器中的内联汇编器不支持 Thumb® 指令集。除非使用 #pragma arm 和 #pragma thumb 编译指示，否则内联汇编器不汇编 Thumb 指令；而且在为 Thumb 状态编译 C 或 C++ 时，内联汇编器根本不能使用 （请参阅第7-6 页的*Thumb 指令集*）。

7.2 嵌入式汇编器

利用 ARM 编译器可将汇编代码外联包括到一个或多个 C 或 C++ 函数定义中。嵌入式汇编器支持对目标处理器进行非受限底层访问，使您可以使用 C 和 C++ 预处理程序指令，还可以方便地对结构成员偏移量进行访问。

有关为 ARM 处理器编写汇编程序的详细信息，请参阅 *《汇编器指南》*。

7.2.1 嵌入式汇编器语法

嵌入式汇编程序定义由 **__asm** （C 和 C++）或 **asm** (C++) 函数限定符标记，可用于：

- 成员函数
- 非成员函数
- 模板函数
- 模板类成员函数。

用 **__asm** 或 **asm** 声明的函数可以有参数并返回一个类型。它们从 C 和 C++ 中调用的方式与普通 C 和 C++ 函数的调用方式相同。嵌入式汇编程序的语法是：

```
__asm return-type function-name(parameter-list)
{
   // ARM/Thumb/Thumb-2 assembler code
   instruction[;instruction]
   ...
   instruction
}
```

如命令行指定的那样，嵌入式汇编器 （ARM 或 Thumb）的初始状态由编译器的初始状态确定。这意味着：

- 如果编译器在 ARM 状态下启动，则嵌入式汇编器将使用 `--arm`
- 如果编译器在 Thumb 状态下启动，则嵌入式汇编器将使用 `--thumb`。

每个函数的起始嵌入式汇编器状态与 `#pragma arm` 和 `#pragma thumb` 编译指示修改的编译器调用时设置相同。

可以在嵌入式汇编器中使用显式 `ARM`、`THUMB` 或 `CODE16` 指令，更改函数内的嵌入式汇编器状态。`__asm` 函数内的此类指令不影响随后的 `__asm` 函数的 ARM 或 Thumb 状态。

如果要编译支持 Thumb-2 的处理器，则可以在 Thumb 状态下使用 Thumb-2 指令。

注意

自变量名允许用在参数列表中，但不能用在嵌入式汇编程序体内。例如，以下函数在函数体内使用整数 i，但在汇编中无效：

```
__asm int f(int i)
{
    ADD i, i, #1 // error
}
```

例如，可以使用 r0 代替 i。

有关 C 和 C++ 源代码中的嵌入式汇编语言的详细信息，请参阅 *《开发指南》*中有关混合使用 C、 C++ 和汇编语言的章节。

嵌入式汇编器示例

示例 7-1 显示了用作嵌入式汇编器例程的字符串复制例程。

示例 7-1 用嵌入式汇编器进行字符串复制

```
#include <stdio.h>
__asm void my_strcpy(const char *src, char *dst)
{
loop
      LDRB  r2, [r0], #1
      STRB  r2, [r1], #1
      CMP   r2, #0
      BNE   loop
      BX    lr
}
int main(void)
{
    const char *a = "Hello world!";
    char b[20];
    my_strcpy (a, b);
    printf("Original string: '%s'\n", a);
    printf("Copied   string: '%s'\n", b);
    return 0;
}
```

7.2.2 嵌入式汇编程序的限制

以下约束适用于嵌入式汇编程序：

- 在预处理之后，__asm 函数只能包含汇编代码，不过以下标识符除外（请参阅第7-21 页的*相关基类的关键字*和第7-22 页的*成员函数类的关键字*）：

```
__cpp(expr)
__offsetof_base(D, B)
__mcall_is_virtual(D, f)
__mcall_is_in_vbase(D, f)
__mcall_offsetof_base(D, f)
__mcall_this_offset(D, f)
__vcall_offsetof_vfunc(D, f)
```

- 编译器不为 __asm 函数生成返回指令。如果要从 __asm 函数返回，必须将用汇编代码编写的返回指令包含在函数体内。

  注意

  因为嵌入式汇编器保证按照已定义的顺序发出 __asm 函数，所以这使得转到下一个函数成为可能。不过，内联和模板函数的行为有所不同（请参阅第7-19 页的*嵌入式汇编函数的生成*）。

- __asm 函数不更改应用的 AAPCS 规则。这意味着，即使 __asm 函数可用的汇编代码（例如，更改状态）中没有限制，在 __asm 函数和普通 C 或 C++ 函数之间的所有调用也必须紧随 AAPCS。

7.2.3 嵌入式汇编表达式和 C 或 C++ 表达式之间的差异

要清楚嵌入式汇编和 C 或 C++ 之间的以下差异：

- 汇编器表达式总是无符号的。相同的表达式在汇编器和 C 或 C++ 中可能有不同值。例如：

```
MOV r0, #(-33554432 / 2)        // result is 0x7f000000
MOV r0, #__cpp(-33554432 / 2)   // result is 0xff000000
```

- 含有前导零的汇编器编号仍是十进制的。例如：

```
MOV r0, #0700                  // decimal 700
MOV r0, #__cpp(0700)           // octal 0700 == decimal 448
```

- 汇编器运算符的优先顺序与 C 和 C++ 不同。例如：

```
MOV r0, #(0x23 :AND: 0xf + 1)     // ((0x23 & 0xf) + 1) => 4
MOV r0, #__cpp(0x23 & 0xf + 1)     // (0x23 & (0xf + 1)) => 0
```

- 汇编器字符串不是空终止的：

```
DCB "Hello world!"               // 12 bytes (no trailing null)
DCB __cpp("Hello world!")        // 13 bytes (trailing null)
```

注意

汇编器规则应用于 __cpp 外部，而 C 或 C++ 规则应用于 __cpp 内部。请参阅第7-20 页的*__cpp 关键字*。

7.2.4 嵌入式汇编函数的生成

在转换单元中，所有 __asm 函数体都进行汇编，就像它们被连接为即将要传递给 ARM 汇编器的单个文件一样。在传递给汇编器的汇编文件中， __asm 函数的顺序保证与源文件中的顺序相同，不过用模板实例生成的函数除外。

注意

这意味着，如果返回指令被忽略，在文件中通过离开第一个函数的末尾进入下一个 __asm 函数来控制从一个 __asm 函数到另一个函数的传递是可能的。

当您调用 armcc 时，生成单个对象文件的局部链接会将汇编器产生的对象文件与编译器的对象文件相结合。

编译器为每个 __asm 函数生成一条 AREA 指令，如示例 7-2 所示：

示例 7-2 __asm 函数

```
#include <cstddef>
struct X
{
    int x,y;
    void addto_y(int);
};
__asm void X::addto_y(int)
{
    LDR      r2, [r0, #__cpp(offsetof(X, y))]
    ADD      r1, r2, r1
    STR      r1, [r0, #__cpp(offsetof(X, y))]
    BX       lr
}
```

对于此函数，编译器生成：

```
    AREA ||.emb_text||, CODE, READONLY
    EXPORT |_ZN1X7addto_yEi|
#line num "file"
|_ZN1X7addto_yEi| PROC
    LDR r2, [r0, #4]
    ADD r1, r2, r1
    STR r1, [r0, #4]
    BX lr
    ENDP
    END
```

offsetof 只能在 __cpp() 内部使用，因为它是 cstddef 头文件中的常规 offsetof 宏。

常规 __asm 函数放在名称为 .emb_text 的 ELF 区域中。也就是说，嵌入式汇编程序从不内联。不过，隐式实例化的模板函数和内联函数的外联副本放在名称源于函数名的区域内，还有一个附加属性将它们标记为公共。这确保这些类型函数的特殊语义得以保持。

注意

由于对内联函数的多个外联副本和模板函数的区域的特殊命名，所以这些函数不按照定义顺序排序，而是任意排序。因此，不能假定代码不执行内联函数或模板函数，而执行另一个 __asm 函数。

7.2.5 __cpp 关键字

使用 **__cpp** 关键字可以在汇编代码中访问 C 或 C++ 编译时常数表达式，包括含有外部链接的数据或函数的地址。__cpp 内的表达式必须是适合用作 C++ 静态初始化的常数表达式。请参阅 ISO/IEC 14882:2003 中的 *3.6.2 非局部对象的初始化 (Initialization of non-local objects)* 和 *5.19 常量表达式 (Constant expressions)*。

示例 7-3 演示一个用于替代 __cpp(*expr*) 的常数：

示例 7-3 __cpp(*expr*)

```
LDR r0, =__cpp(&some_variable)
LDR r1, =__cpp(some_function)
BL  __cpp(some_function)
MOV r0, #__cpp(some_constant_expr)
```

__cpp 表达式中的名称是在 __asm 函数的 C++ 上下文中查找的。__cpp 表达式结果中的任何名称按照要求被损毁，并自动为其生成 IMPORT 语句。

7.2.6 手动解决重载

示例 7-4 演示使用 C++ 类型转换为非虚拟函数调用解决重载：

示例 7-4 C++ 转换

```
void g(int);
void g(long);
struct T
{
    int mf(int);
    int mf(int,int);
};
__asm void  f(T*, int, int)
{
    BL __cpp(static_cast<int (T::*)(int, int)>(&T::mf)) // calls T::mf(int, int)
    BL __cpp(static_cast<void (*)(int)>(g)) // calls g(int)
    BX lr
}
```

7.2.7 相关基类的关键字

利用以下关键字，可以确定从对象起始处到其中的基类子对象的偏移量：

__offsetof_base(D, B)

B 必须是 D 的明确的非虚基类。

返回从 D 对象的起始处到其中 B 基子对象的起始处的偏移量。结果可能为零。示例 7-5 显示必须添加到 D* p 以实现 static_cast<B*>(p) 的等效功能的偏移量 （以字节为单位）。

示例 7-5 static_cast<B*>(p)

```
__asm B* my_static_base_cast(D* /*p*/)
{
    if __offsetof_base(D, B) <> 0 // optimize zero offset case
        ADD r0, r0, #__offsetof_base(D, B)
```

```
    endif
    BX lr
}
```

在汇编器源代码中，这些关键字将转换为整数或逻辑常数。只能将它们用于 __asm 函数，而不能用于 __cpp 表达式。

7.2.8 成员函数类的关键字

以下关键字方便了从 __asm 函数中调用虚拟成员函数和非虚拟成员函数。以 __mcall 开头的关键字可用于虚拟函数和非虚拟函数。以 __vcall 开头的关键字只能用于虚拟函数。在调用静态成员函数的过程中，这些关键字没有特别的帮助。

有关如何使用这些关键字的示例，请参阅第7-23 页的*调用非静态成员函数*。

__mcall_is_virtual(D, f)

如果 f 是 D 中的虚拟成员函数或是 D 的基类，则结果为 {TRUE}，否则为 {FALSE}。如果返回 {TRUE}，则可以使用虚拟调度进行调用，否则必须直接进行调用。

__mcall_is_in_vbase(D, f)

如果 f 是 D 的虚基类中的非静态成员函数，则结果为 {TRUE}，否则为 {FALSE}。如果返回 {TRUE}，则 this 调整必须使用 __mcall_offsetof_vbase(D, f) 完成，否则必须使用 __mcall_this_offset(D, f) 完成。

__mcall_offsetof_vbase(D, f)

其中， D 是类类型， f 是 D 的虚基类中定义的非静态成员函数，换言之， __mcall_is_in_vbase(D,f) 返回 True。

在保持基偏移的 vtable 槽的 vtable 中有负偏移时返回这种结果（从 D 对象的起始处到定义 f 的基的起始处）。

在使用指向 D 的指针调用 f 的过程中，这是必要的 this 调整。

注意

偏移返回一个正值，随后要从 vtable 指针减去该值。

__mcall_this_offset(D, f)

其中， D 是类类型， f 是 D 中定义的非静态成员函数或 D 的非虚基类。

返回从 D 对象起始处到定义 f 的基的起始处的偏移量。在使用指向 D 的指针调用 f 的过程中，这是必要的 this 调整。如果在 D 中找到 f，则为零，否则与 __offsetof_base(D,B) 相同，其中， B 是 f 所在的 D 的非虚基类。

在 D 的虚拟基类中找到 f 时，如果使用 __mcall_this_offset(D,f)，则返回任意值，使用该值时会导致汇编错误。正是这样，在将要跳过的汇编代码部分中会发生对 __mcall_this_offset 的此类无效使用。

__vcall_offsetof_vfunc(D, f)

其中 D 是类， f 是 D 中定义的虚拟函数或是 D 的基类。

函数返回保持基偏移的 vtable 中的槽的负偏移。基偏移的计算结果为 D 对象到定义 f 的基的起始处之间的距离。

当 f 不是虚拟成员函数时，如果使用 __vcall_offsetof_vfunc(D, f)，则返回任意值，使用该值时会导致汇编错误。

7.2.9 调用非静态成员函数

可以使用以 __mcall 和 __vcall 开头的关键字在 __asm 函数中调用非虚拟函数和虚拟函数。请参阅第7-22 页的*成员函数类的关键字*。静态成员函数的参数不相同（也就是说，没有 this），造成没有 __mcall_is_static 检测静态成员函数，因此调用位置很可能已经专用于调用静态成员函数。

调用非虚拟成员函数

示例 7-6 所示的代码可用于调用虚基类或非虚基类中的非虚拟函数：

示例 7-6 调用非虚拟函数

```
// rp contains a D* and we want to do the equivalent of rp->f() where f is a
// nonvirtual function
// all arguments other than the this pointer are already setup
// assumes f does not return a struct
 if __mcall_is_in_vbase(D, f)
   LDR r12, [rp]                                    // fetch vtable pointer
   LDR r0, [r12, #-__mcall_offsetof_vbase(D, f)]    // fetch the vbase offset
   ADD r0, r0, rp                                   // do this adjustment
 else
   ADD r0, rp, #__mcall_this_offset(D, f)           // set up and adjust this
```

```
                                                    // pointer for D*
 endif
   BL __cpp(&D::f)                                  // call D::f
```

调用虚成员函数

示例 7-7 所示的代码可用于调用虚基类或非虚基类中的虚拟函数：

示例 7-7 调用虚拟函数

```
// rp contains a D* and we want to do the equivalent of rp->f() where f is a
// virtual function
// all arguments other than the this pointer are already setup
// assumes f does not return a struct
 if __mcall_is_in_vbase(D, f)
    LDR r12, [rp]                                   // fetch vtable pointer
    LDR r0, [r12, #-__mcall_offsetof_vbase(D, f)] // fetch the base offset
    ADD r0, r0, rp                                  // do this adjustment
    LDR r12, [r0]                                   // fetch vbase vtable pointer
 else
    MOV r0, rp                                      // set up this pointer for D*
    LDR r12, [rp]                                   // fetch vtable pointer
    ADD r0, r0, #__mcall_this_offset(D, f)          // do this adjustment
 endif
    MOV lr, pc                                      // prepare lr
    LDR pc, [r12, #__vcall_offsetof_vfunc(D, f)]  // calls rp->f()
```

7.3 访问 sp、 lr 或 pc 的旧内联汇编器

ARM Developer Suite (ADS) v1.2 及更早版本中的编译器可以从内联汇编代码中访问 sp (r13)、lr (r14) 和 pc (r15)（请参阅第7-2 页的*内联汇编器*）。示例 7-8 演示旧内联汇编代码如何使用 lr。

示例 7-8 使用 lr 的旧内联汇编代码

```
void func()
{
    int var;
    __asm
    {
        mov  var, lr  /* get the return address of func() */
    }
}
```

如果旧代码在内联汇编中使用函数的返回地址，则无法保证 lr 包含该地址。例如，某些编译选项或优化可能将 lr 用于其他用途。如果以这种方式使用 lr、sp 或 pc，则 RVCT v2.0 和更高版本中的编译器会报告类似以下的错误：

如果要从 C 或 C++ 源文件内访问这些寄存器，您可以：

- 使用嵌入式汇编程序 （请参阅第7-16 页的*嵌入式汇编器*）。
- 在内联汇编中使用以下内在函数：

  `__current_pc()` 访问 pc 寄存器。

  `__current_sp()` 访问 sp 寄存器。

  `__return_address()` 访问 lr 寄存器。

另请参阅：

- 第7-26 页的*访问旧代码中的 sp (r13)、 lr (r14) 和 pc (r15)*
- *《编译器参考指南》*中第4-71 页的*指令内在函数*。

7.3.1 访问旧代码中的 sp (r13)、 lr (r14) 和 pc (r15)

通过以下方法，可以在源代码中正确访问 sp、 lr 和 pc 寄存器：

方法 1 在内联汇编中使用编译器内在函数，例如：

```
void printReg()
{
    unsigned int spReg, lrReg, pcReg;
    __asm
    {
        MOV spReg, __current_sp()
        MOV pcReg, __current_pc()
        MOV lrReg, __return_address()
    }
    printf("SP = 0x%X\n",spReg);
    printf("PC = 0x%X\n",pcReg);
    printf("LR = 0x%X\n",lrReg);
}
```

方法 2 使用嵌入式汇编从 C 或 C++ 源文件内访问物理 ARM 寄存器，例如：

```
__asm void func()
{
    MOV r0, lr
    ...
    BX lr
}
```

通过这种方式可以捕获和显示函数的返回地址，例如，出于调试目的而显示调用树。

请参阅第7-16 页的*嵌入式汇编器*。

——注意——

编译器也可将函数内联到其调用方函数。如果一个函数已内联，则返回地址是调用该内联函数的函数的返回地址。同样，函数可以是尾调用。

请参阅 *《编译器参考指南》*中第4-89 页的*__return_address*。

7.4 内联汇编代码与嵌入式汇编代码之间的差异

内联汇编和嵌入式汇编的编译方法有所不同：

- 内联汇编代码使用高级处理器分离，并在代码生成过程中与 C 和 C++ 代码集成。因此，编译器将 C 和 C++ 代码与汇编代码一起进行优化。
- 与内联汇编代码不同，嵌入式汇编代码与 C 和 C++ 代码分开进行汇编，并生成一个编译对象，该编译对象随后与 C 或 C++ 源代码的编译对象相结合。
- 可通过编译器来内联内联汇编代码，但无论是显式还是隐式，都无法内联嵌入式汇编代码。

表 7-1 汇总了内联汇编器与嵌入式汇编器之间的主要差异。

表 7-1 内联汇编器与嵌入式汇编器之间的差异

功能	嵌入式汇编器	内联汇编器
指令集	ARM 和 Thumb。	仅 ARM。
ARM 汇编器指令	全部支持。	不支持。
ARMv6 指令	全部支持。	支持大多数指令，不过有一些例外，例如 SETEND 和一些系统扩展。支持完整的 ARMv6 SIMD 指令集。
ARMv7 指令	全部支持。	不支持。
C/C++ 表达式	仅常数表达式。	完整 C/C++ 表达式。
优化汇编代码	不优化。	全部优化。
内联	从不。	可能。
寄存器访问	使用指定的物理寄存器。还可以使用 PC、LR 和 SP。	使用虚拟寄存器（请参阅第7-8 页的*虚拟寄存器*）。使用 sp (r13)、lr (r14) 和 pc (r15) 则会产生错误。请参阅第7-25 页的*访问 sp、lr 或 pc 的旧内联汇编器*。
返回指令	必须将其添加到代码中。	自动生成。（不支持 BX、BXJ 和 BLX 指令。）
BKPT 指令	直接支持。	不支持。

另请参阅第7-18 页的*嵌入式汇编表达式和 C 或 C++ 表达式之间的差异*。