RealView® 编译工具

4.0 版

库和浮点支持指南

ARM®

ARM DUI 0349BC

RealView 编译工具
库和浮点支持指南

版本信息

本手册进行了以下更改。

更改历史记录

日期	发行号	保密性	变更
2007 年 3 月	A	非保密	RealView Development Suite v3.1 3.1 版
2008 年 9 月	B	非保密	RealView Development Suite v4.0 4.0 版

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
库和浮点支持指南

前言

本前言介绍 《RealView 编译工具库和浮点支持指南》。本前言分为以下几节：

关于本手册

本手册介绍了 ARM C 和 C++ 库、与 ISO 标准的符合情况、与目标相关的函数调整以及应用程序特有的要求。本手册还介绍了 ARM C 微型库和 ARM 的浮点计算支持。

适用对象

本手册是为所有使用 *RealView 编译工具* (RVCT) 构建应用程序的开发人员编写的，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

本手册由以下章节组成：

第 1 章 *简介*

本章简要介绍了 ARM C 和 C++ 库以及 ARM 浮点环境。

第 2 章 *C 和 C++ 库*

本章介绍了 ARM C 和 C++ 库，并提供了有关重新实现各个库函数的说明。ARM C 库提供了附加组件以实现 C++ 支持，并为不同体系结构和处理器编译代码。

有关 Rogue Wave 标准 C++ 库的说明，请参阅随 RVCT 提供的 Rogue Wave HTML 文档。

第 3 章 *C 微型库*

本章介绍了 ARM C 微型库，并提供了有关重新实现各个微型库函数的说明。

第 4 章 *浮点支持*

本章介绍了 ARM 编译器和库中的浮点支持。

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*。例如， *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：

- 《RVDS 入门指南》 (ARM DUI 0255)
- 《RVCT 要点指南》 (ARM DUI 0202)
- 《RVCT 编译器用户指南》 (ARM DUI 0205)
- 《RVCT 编译器参考指南》 (ARM DUI 0348)
- 《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)

本手册提供了一个术语表。请参阅 《RVDS 入门指南》。

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

此外，有关与 ARM 产品相关的特定信息，请参阅下列文档：

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

其他出版物

本文中引用了下列出版物：

- IEEE 754 - 1985 IEEE 二进制浮点算法标准

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

本章简要介绍了 ARM C 和 C++ 库以及 ARM 浮点环境。本章分为以下几节：

- 第1-2 页的*关于运行时库*
- 第1-7 页的*关于浮点支持*

1.1 关于运行时库

提供的下列运行时库用于支持编译的 C 和 C++：

C standardlib

C 库包含：

- ISO C 库标准定义的函数。
- 与目标相关的函数，用于在半主机执行环境中实现 C 库函数。您可以在自己的应用程序中重新定义这些函数。
- 辅助函数，用于在 C 和 C++ 中进行编译。

C microlib

C 微型库 (microlib) 包含：

- 高度优化以最大限度减少代码大小的函数。
- 不符合 ISO C 库标准的函数。
- 不符合 IEEE 754 二进制浮点算法标准的函数。

C++

C++ 库包含 ISO C++ 库标准定义的函数。

C++ 库依靠 C 库获得目标特定的支持。C++ 库中不存在目标相关性。

C++ 库包含：

- Rogue Wave 标准 C++ 库
- 编译 C++ 时使用的辅助函数
- Rogue Wave 库不支持的附加 C++ 函数。

注意

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

1.1.1 ARM 体系结构 ABI 遵从性

ARM 体系结构的*应用程序二进制接口* (ABI) 是一个规范系列，描述了有关将源程序转换为对象文件的处理器特定要求。通过将符合 ABI 相关要求的任何工具链生成的对象文件链接在一起，可以生成一个最终可执行映像或库。规范中的每个文档介绍了特定领域的兼容性。例如，《ARM 体系结构的 C 库 ABI》 (*C Library ABI for the ARM Architecture*) (CLIBABI) 介绍了对所有符合标准的实现来说都相同的那部分 C 库。ABI 文档包含几个标记为*平台特定的*领域。要定义完整执行环境，您必须提供这些平台特定的细节。这产生了一些补充规范，如《ARM GNU/Linux ABI 补充规范》。通过 《ARM 体系结构的基本标准 ABI》 (*Base Standard ABI for the ARM Architecture*) (BSABI)，您可以使用不同开发商编写的 ARM、Thumb 以及 Thumb-2 对象和库（均支持 ARM 体系结构的 ABI）。RVDS 完全支持 BSABI，其中包括对 DWARF 3 调试表 （DWARF 调试标准第 3 版）的支持。

有关 ARM 支持的基本标准、其他 *ARM 嵌入式 ABI* (AEABI)、软件接口以及其他标准的完整信息，请参阅 *install_directory*\Documentation\Specifications\...。

有关最新发行版本的详细信息，请访问 http://www.arm.com。

ARM C 和 C++ 库符合 BSABI 和以下 AEABI 中介绍的标准：

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

测试一致性

如果需要完整的 CLIBABI 可移植性，请在使用 #include 包含任何库头之前指定 #define _AEABI_PORTABILITY_LEVEL 1，例如 <stdlib.h>。也可以在命令行中使用 -D_AEABI_PORTABILITY_LEVEL=1 来实现此目的。这可提高您的对象文件到其他 *CLIBABI* 实现的可移植性，但会降低某些库运算的性能。

有关详细信息，请参阅 *install_directory*\Documentation\Specifications\... 中的 CLIBABI 规范 clibabi.pdf。

1.1.2 库目录结构

库安装在 RVCT 库目录 ...\lib 下面的两个子目录中：

armlib　包含 ARM C 库变体、浮点算法库 (fplib) 和数学库 (mathlib)。附带的头文件位于 ...\include 中。

cpplib　包含 Rogue Wave C++ 库的变体 (cpp_*) 和支持的 ARM C++ 函数 (cpprt_*)，它们统称为 ARM C++ 库。附带的头文件安装在 ...\include 中。

必须将 RVCT40LIB 环境变量设置为指向 lib 目录；或者，如果未设置此变量，则必须指向 ARMLIB。此外，也可以使用链接器的 --libpath 参数来指定包含库子目录的目录。不能单独指定 armlib 和 cpplib 目录，因为在将来的版本中这种目录结构可能会发生变化。链接器将从 lib 位置中查找它们。

注意

- 仅以二进制格式提供 ARM C 库。
- 不能对 ARM 库进行修改。如果要创建新的库函数实现，应将新函数放在对象文件或您自己的库中，并在链接应用程序时包括该函数。将使用您创建的函数版本，而不是标准库版本。
- 通常， ISO C 库中只有少数几个函数需要重新实现来创建与目标相关的应用程序。
- Rogue Wave 标准 C++ 库源代码不是免费分发的。可以从 Rogue Wave Software Inc. 或通过 ARM Limited 获得源代码，但需要支付额外的许可费用。有关详细信息，请参阅 *install_directory*\Documentation\RogueWave 中的 Rogue Wave 在线文档。

1.1.3 构建选项和库变体

在构建您自己的应用程序时，必须在某些方面做出选择。例如：

目标体系结构和指令集

ARM、 Thumb® 或 Thumb-2。

字节顺序 大端或小端。

浮点支持 软件 (SoftVFP)、硬件 (VFP)、具有半精度或双精度扩展或不支持浮点的软件或硬件。

与位置无关

数据可以是读/写型，且可与位置无关或相关。

代码可以是只读型的，且可与位置无关或相关。

注意

microlib 不支持位置无关性。

在链接汇编器代码、 C 或 C++ 代码时，链接器将选择与指定的生成选项兼容的相应 C 和 C++ 库变体。每个主要生成选项组合都有一个 ISO C 库变体。

有关详细信息，请参阅：

- 《实用程序指南》中第 3 章 *使用 armar*
- 《编译器用户指南》中第2-23 页的*指定过程调用标准 (AAPCS)*
- 《汇编器指南》

1.1.4 使用 VFP 支持库

VFP 支持库由 VFP 支持代码使用。VFP 支持代码是从未定义的指令捕获中执行的，在发生异常浮点情况时将触发该捕获。

示例代码是在主示例目录的 `...\vfpsupport` 中提供的。此代码说明了如何设置未定义的指令处理程序来调用 VFP 支持代码。

1.1.5 Thumb C 库

如果链接器检测到为以下内容构建了一个或多个要链接的对象，它将自动在 Thumb C 库中进行链接：

- Thumb 或 Thumb-2 （使用 `--thumb` 选项或 `#pragma thumb`）
- 交互操作 （在体系结构 ARMv4T 上使用 `--apcs /interwork` 选项）
- ARM v6-M 体系结构目标或处理器，例如 Cortex-M1
- ARMv7-M 体系结构目标或处理器，如 Cortex-M3。

无论使用什么名称， Thumb C 库可能并非只包含 Thumb 代码。如果 ARM 指令可用，则 Thumb 库可能会使用这些指令提高关键函数 （如 `memcpy()`、`memset()` 和 `memclr()`）的性能。不过，为了最大限度地减少代码大小，Thumb C 库的大部分都是用 Thumb 编码的。

对于纯 ARM 生成，请使用 `--arm_only` 选项进行编译。

注意

用于 ARMv7-M 目标的 Thumb C 库仅包含 Thumb-2 代码。

有关详细信息，请参阅第 2 章 *C 和 C++ 库*。

1.2 关于浮点支持

ARM 浮点环境实现了 IEEE 754 二进制浮点算法标准。有关 ARM 实现的这一标准的详细信息，请参阅第4-35 页的*IEEE 754 算法*。

ARM 系统可能具有：

- VFP 协处理器
- 没有浮点硬件。

如果为具有硬件 VFP 协处理器的系统进行编译，ARM 编译器将使用协处理器。如果为没有协处理器的系统进行编译，编译器将在软件中执行计算。

例如，编译器选项 `--fpu=vfp` 选择一个硬件 VFP 协处理器；而选项 `--fpu=softvfp` 指定在软件中执行算术运算，而无需使用任何协处理器指令。

有关详细信息，请参阅第 4 章 *浮点支持*。

第 2 章
C 和 C++ 库

本章介绍了 C 和 C++ 库。这些库支持使用 C 或 C++ 编写的程序。本章分为以下几节：

2.1 关于 C 和 C++ 库

本节包含有关 C 和 C++ 库的信息。如果您要编写需要装入到极少量内存中的深层嵌入式应用程序，请参阅第 3 章 *C 微型库*。

2.1.1 C 和 C++ 库功能

在 RVCT 中， C 库使用标准 ARM 半主机环境来提供文件输入/输出等功能。*RealView ARMulator® ISS*、 RealView ICE、*实时仿真器模型* (RTSM) 和*指令集系统模型* (ISSM) 均支持此环境。有关调试环境的详细信息，请参阅 《开发指南》中的第 8 章 *半主机*。

可以将 C 库中任何与目标相关的函数作为您的应用程序的一部分进行重新实现。这样，即可根据您自己的执行环境调整 C 库，进而也可以对 C++ 库进行调整。

还可以调整很多与目标无关的函数以满足您自己的应用程序特有要求，例如：

- `malloc` 系列
- `ctype` 系列
- 所有语言环境特有的函数。

许多 C 库函数独立于所有其他函数，并且不包含目标相关性。可以方便地从汇编器代码中使用这些函数。

2.1.2 命名空间

所有 C++ 标准库名称都是在命名空间 `std` 中使用以下 C++ 语法定义的，其中包括 C 库名称 （如果包含它们）。

```
#include <cstdlib> // instead of stdlib.h
```

这意味着，必须使用以下方法之一限定所有库名称。

- 指定标准命名空间，例如：
  ```
  std::printf("example\n");
  ```
- 使用 C++ 关键字 **`using`** 向全局命名空间导入一个名称：
  ```
  using namespace std;
  printf("example\n");
  ```
- 使用编译器选项 `--using_std`。

注意

errno 是一个宏，因此，不必使用命名空间对其进行限定。

2.2 编写可重入且线程安全的代码

ARM C 库支持多线程，例如，在使用*实时操作系统* (RTOS) 时。在本节的其余部分中介绍多线程时，将使用以下定义：

线程 表示多个彼此之间共享全局数据的执行流。

进程 表示共享一组特定全局数据的所有线程的集合。

如果计算机中存在多个进程，它们可能是完全独立的，因而不共享任何数据（特殊情况除外）。类似地，每个进程可以是单线程进程，也可以分为多个线程。

如果是单线程进程，则只有一个控制流。不过，在多线程应用程序中，几个控制流可能会试图同时访问相同的函数和资源。要保护资源的完整性，为多线程应用程序编写的所有代码都必须是*可重入*且*线程安全的*。

2.2.1 可重入性和线程安全性简介

可重入性和线程安全性均与函数处理资源的方式有关。但是，它们是不同的：

- 可重入函数既不会在连续调用中存储静态数据，也不会返回指向静态数据的指针。对于这种类型的函数，调用方将提供函数所需的所有数据，如指向任何工作区的指针。这意味着，函数的多个并发调用不会相互干扰。

  注意

  可重入函数不能调用非可重入函数。

- 线程安全函数使用*锁*保护共享资源，以防止对其进行并发访问。线程安全性只涉及函数实现方式，而不涉及其外部接口。在 C 中，局部变量是在堆栈上动态分配的。因此，任何不使用静态数据或其他共享资源的函数通常都是线程安全的。

2.2.2 在 C 库中使用静态数据

静态数据是指没有存储在堆栈或堆中的永久性读/写数据。此永久性数据的范围可以是外部或内部，并且：

- 位于固定地址 （使用 `--apcs /norwpi` 编译时）
- 位于相对于静态基址寄存器 `r9` 的固定地址 （使用 `--apcs /rwpi` 编译时）。

使用静态数据的库可以是可重入的，但这取决于其是否使用 __user_libspace 静态数据区和选择的构建选项：

- 使用 --apcs /norwpi 编译时，将按与位置相关的方式确定读/写静态数据的地址。这是缺省设置。这些变体的代码是单线程的，因为它使用读/写静态数据。
- 使用 --apcs /rwpi 编译时，将使用静态基址寄存器 sb 的偏移，按与位置无关的方式确定读/写静态数据的地址。这些变体的代码是可重入的，并且可能是多线程的 （如果每个线程使用不同的静态基址值）。

下面介绍了 C 库如何使用静态数据：

- 缺省浮点算法库 fz_* 和 fj_* 不使用静态数据，并且始终是可重入的。但是， f_* 和 g_* 库使用静态数据。
- C 库中的所有静态初始化数据都是只读的。
- 所有可写的静态数据都是零初始化的。
- 大多数 C 库函数不使用可写的静态数据；无论是使用缺省构建选项 --apcs /norwpi 还是可重入构建选项 --apcs /rwpi 进行构建，这些库函数都是可重入的。
- 某些函数在其定义中包含隐式静态数据。除非使用 --apcs /rwpi 生成可重入应用程序，并且调用方在 sb 中使用不同的值，否则不能在该应用程序中使用这些函数。

有关此处介绍的 --apcs 生成选项的信息，请参阅 《编译器用户指南》中第2-23 页的*位置无关限定符*。

——注意——

在将来的版本中，在定义中使用静态数据的具体函数可能会发生变化。

2.2.3 __user_libspace 静态数据区

__user_libspace 静态数据区保存 C 库的静态数据。这是一个由 C 库提供的零初始化数据块 （96 个字节）。在 C 库初始化期间，还会将其用作临时堆栈。

缺省 ARM C 库使用 __user_libspace 区保存以下内容：

- errno，由可设置 errno 的任何函数使用。缺省情况下， __rt_errno_addr() 返回一个指向 errno 的指针。

- 软件浮点的 FP 状态字 （异常标记、舍入模式）。硬件浮点中不使用此状态字。缺省情况下， __rt_fp_status_addr() 返回一个指向 FP 状态字的指针。
- 指向堆基址的指针 （即， __Heap_Descriptor），由与 malloc 相关的所有函数使用。
- alloca 状态，由 alloca() 及其辅助函数使用。
- 当前语言环境，由 setlocale() 等函数使用，也供依赖于它们的所有其他库函数使用。例如， ctype.h 函数需要访问 LC_CTYPE 设置。

C++ 库使用 __user_libspace 区保存以下内容：

- new_handler 字段和 ddtor_pointer：
  - new_handler 字段用于跟踪传递给 std::set_new_handler() 的值
  - ddtor_pointer 由 __cxa_atexit() 和 __aeabi_atexit() 使用。
- std::set_terminate() 和 std::set_unexpected() 等函数的 C++ 异常处理信息。

请参阅下列规范：有关 __aeabi_atexit()、 std::set_terminate() 和 std::set_unexpected() 的详细信息，请参阅 *CPPABI* 和 《ARM 体系结构的异常处理 ABI》 (*Exception Handling ABI for the ARM Architecture*)。

注意

在将来的版本中， C 和 C++ 库使用 __user_libspace 区的方式可能会发生变化。

确定 __user_libspace 的地址

以下两个包装函数用于返回 __user_libspace 静态数据区的小节：

__user_perproc_libspace()

返回一个指向 96 个字节的数据块的指针，该数据块用于存储整个进程中的全局数据，即，在所有线程之间共享的数据。

__user_perthread_libspace()

返回一个指向 96 个字节的数据块的指针，该数据块用于存储特定线程的局部数据。这意味着， __user_perthread_libspace() 根据从中调用它的线程返回不同的地址。

重新实现 __user_libspace

通常，不需要重新定义 __user_libspace() 函数。但是，如果编写的是操作系统或进程切换程序，则必须重新实现此函数。有关详细信息，请参阅第2-40 页的*调整静态数据访问*。

将单线程进程移植到重新实现了该函数的 RVCT 中时，您可以继续使用该函数，而无需更改代码。但是，如果使用 RVCT 编写多线程应用程序，则更改行为是非常重要的。

2.2.4 在多线程应用程序中管理锁

线程安全函数使用锁保护共享资源，以防止对其进行并发访问。可以重新实现一些函数以便能够管理锁定机制，从而防止共享数据 （如堆）损坏。这些函数具有下列原型：

_mutex_initialize()

```
int _mutex_initialize(mutex *m);
```

此函数接受指向 32 位字的指针，并将它作为有效的互斥量来初始化。

缺省情况下， _mutex_initialize() 为非线程应用程序返回零。因此，在多线程应用程序中， _mutex_initialize() 必须在成功时返回非零值，以使库在运行时知道它正在多线程环境中使用。

如果使用的是互斥量，则必须提供此函数。

_mutex_acquire()

```
void _mutex_acquire(mutex *m);
```

此函数使调用线程在提供的互斥量上获得锁。

如果互斥量没有所有者， _mutex_acquire() 则会立即返回。如果互斥量归另一个线程所有， _mutex_acquire() 必须将其阻止，直至互斥量变为可用。

如果使用的是互斥量，则必须提供此函数。

_mutex_release()

```
void _mutex_release(mutex *m);
```

此函数使调用线程在 _mutex_acquire() 所获取的互斥量上释放锁。

该互斥量仍然存在，并可由随后对 mutex_acquire() 的调用重新锁定。

_mutex_release() 假定互斥量归调用线程所有。

如果使用的是互斥量，则必须提供此函数。

_mutex_free()

```
void _mutex_free(mutex *m);
```

此函数导致调用线程释放提供的互斥量。与该互斥量关联的所有操作系统资源都将被释放。该互斥量将被销毁，不能重用。

_mutex_free() 假定互斥量归调用线程所有。

此函数是可选的。如果未提供此函数， C 库就不会尝试调用它。

对于 C 库，互斥量被指定为可放在任何位置的单个 32 位内存字。但是，如果互斥量实现需要比 32 位内存字更多的空间，或者要求互斥量位于特殊内存区域中，则必须将缺省互斥量视为指向真实互斥量的指针。

从 ARM C 库函数中对互斥量函数进行的调用是弱调用。这意味着在链接代码时，不会搜索库来解析这些符号，即使在命令行显式提供了用户库也是如此。此外，这还表明在此链接失败时不会生成错误消息。从库中进行的调用将变为 NOP 指令。

通过使用下列方法之一，可确保互斥量函数正常工作：

- 将互斥量函数放在非库对象文件中。这有助于确保它们在链接时得到解析。
- 将互斥量函数放在库对象文件中，但前提是除了 C 库中的弱引用之外，还为该目标中的某些内容安排了非弱引用。
- 让链接器从库中显式提取一个特定目标，并将其视为命令行中提及的独立对象文件。通过在调用链接器时写入 libraryname.a(objectfilename.o)，可以实现此方法。

此外，若要确保链接器不会因在链接时将互斥量函数视为未使用而将其删除，请使用 --keep 链接器选项，或在其中存在对其他某些内容的非弱引用的节中定义互斥量函数。

（在 ELF 对象文件中，未定义的符号可能会标有名为 weak 的标记。此类目标将被描述为对该符号进行弱引用。同样，执行同样的功能但没有此标记的目标则进行非弱引用。）

2.2.5 将 ARM C 库用于多线程应用程序

要在多线程环境中使用 ARM 库，您必须提供：

- `__user_perthread_libspace()` 实现，它为每个线程返回不同的内存块。可使用以下任一方法来实现此目的：
  - 根据从中调用它的线程返回不同的地址
  - 将单个 `__user_perthread_libspace` 块放在固定地址，并在切换线程时交换其内容。

  根据您的环境，可使用上述任一方法。

  除非有特别原因，否则，不需要重新实现 `__user_perproc_libspace`。在大多数情况下，不需要重新实现该函数。
- 管理多个堆栈的方法。

  一种执行此操作的简便方法是使用 ARM 双区内存模型 （请参阅第2-67 页的*内存模型*）。使用此模型意味着，将属于主线程的堆栈与堆完全分开。以后，必须从堆本身中为额外堆栈分配更多的内存。
- 线程管理函数，如用于创建或销毁线程、处理线程同步以及检索退出代码。

  ——注意——

  ARM C 库并不提供其自己的线程管理函数，因此，您必须提供所需的任何函数。
- 线程切换机制。

  ——注意——

  ARM C 库并不提供其自己的线程切换机制。这是因为，可以使用多种不同的方法实现此目的，这些库旨在与所有这些方法配合使用。

当需要调用互斥量函数时，只需提供其实现即可。请参阅第2-8 页的*在多线程应用程序中管理锁*。

在某些应用程序中，互斥量函数可能没有什么用处。例如，只要协作线程程序在关键节中避免调用其生成的函数，它就无需采取任何措施来确保数据完整性。不过，在其他类型的应用程序中，这些函数在处理锁方面具有重要作用，例如，在实现优先调度的应用程序或*对称多处理器* (SMP) 模型中。

如果已满足所有这些要求，则可以在多线程环境中使用 ARM C 库。其中：

- 某些函数在每个线程中独立工作

- 某些函数自动使用互斥量函数协调对共享资源的多个访问
- 某些函数仍然是不可重入的，因而提供了等效可重入函数
- 有几个函数仍然是非可重入的，并且没有提供替代函数。

有关详细信息，请参阅*ARM C 库中的线程安全性*和第2-17 页的*ARM C++ 库中的线程安全性*。

2.2.6 ARM C 库中的线程安全性

在 ARM 库中，函数可能是线程安全的，如下所示：

- 某些函数从来都不是线程安全的，例如 `setlocale()`
- 某些函数在本质上就是线程安全的，例如 `memcpy()`
- 某些函数 （例如 `malloc()`）可通过实现 `_mutex_*` 函数变为线程安全的函数
- 其他函数仅在传递了适当参数时才是线程安全的，例如 `tmpnam()`。

如果应用程序以隐藏方式使用 ARM 库 （如使用语言辅助函数），则可能会出现线程问题。

线程安全的函数

表 2-1 显示了线程安全的 C 库函数。

表 2-1 线程安全的函数

函数	说明
calloc(), free(), malloc(), realloc()	如果实现了 _mutex_* 函数，则堆函数是线程安全的。 在所有线程之间共享单个堆，并使用互斥量以避免进行并发访问时发生数据损坏。每个堆实现都负责进行自己的锁定。如果您提供了自己的分配器，它也必须进行自己的锁定。这样，它便可进行精细锁定 （如果需要），而不是简单地使用单个互斥量保护整个堆（粗放锁定）。
__alloca(), __alloca_finish(), __alloca_init(), __alloca_initialize()	如果实现了 _mutex_* 函数，则 alloca 函数是线程安全的。 每个线程的 alloca 状态包含在 __user_perthread_libspace 块中。这意味着多个线程不会发生冲突。 **注意** 请注意， alloca 函数也使用堆。不过堆函数都是线程安全的。
abort(), raise(), signal(), fenv.h	ARM 信号处理函数和 FP 异常捕获是线程安全的。 信号处理程序和 FP 捕获设置是整个进程中的全局设置，并使用锁对其进行保护。这样，即使多个线程同时调用 signal() 或 fenv.h 函数，也不会损坏数据。但要注意，调用影响所有线程，而不是只影响调用线程。

表 2-1 线程安全的函数 （续）

函数	说明
clearerr(), fclose(), feof(),ferror(), fflush(), fgetc(),fgetpos(), fgets(), fopen(),fputc(), fputs(), fread(),freopen(), fseek(), fsetpos(),ftell(), fwrite(), getc(),getchar(), gets(), perror(),putc(), putchar(), puts(),rewind(), setbuf(), setvbuf(),tmpfile(), tmpnam(), ungetc()	如果实现了 _mutex_* 函数，则 stdio 库是线程安全的。 每个单独的流都使用锁进行保护，因此，两个线程可以分别打开并使用其自己的 stdio 流，而不会相互干扰。 如果两个线程都要读取或写入相同的流，fgetc() 和 fputc() 级别的锁定可防止发生数据损坏，但是，每个线程的单独字符输出可能会交叉出现，因而容易造成混淆。 **注意** 请注意，tmpnam() 也包含一个静态缓冲区，但仅在自变量为 NULL 时才使用它。要确保 tmpnam() 使用是线程安全的，应提供您自己的缓冲区空间。
fprintf(), printf(), vfprintf(), vprintf(), fscanf(), scanf()	使用这些函数时： • 标准 C printf() 和 scanf() 函数使用 stdio，因而是线程安全的。 • 如果在多线程程序中调用标准 C printf()，其语言环境可能会发生变化。
clock()	clock() 包含程序静态数据，此数据是在启动时一次性写入的，以后只能对其进行读取。因此，clock() 是线程安全的，但前提是在初始化库时没有运行任何其他线程。
errno()	errno 是线程安全的。 每个线程将其自己的 errno 存储在 __user_perthread_libspace 块中。这意味着，每个线程可以单独调用 errno 设置函数，然后检查 errno，而不用担心受其他线程的影响。
atexit()	atexit() 维护的退出函数列表是进程全局性的，并且使用锁对其进行保护。 在最坏的情况下，如果多个线程调用 atexit()，则不能保证调用退出函数的顺序。

表 2-1 线程安全的函数 （续）

函数	说明
abs(), acos(), asin(),atan(), atan2(), atof(),atol(), atoi(), bsearch(),ceil(), cos(), cosh(),difftime(), div(), exp(),fabs(), floor(), fmod(),frexp(), labs(), ldexp(),ldiv(), log(), log10(),memchr(), memcmp(), memcpy(),memmove(), memset(), mktime(),modf(), pow(), qsort(),sin(), sinh(), sqrt(),strcat(), strchr(), strcmp(),strcpy(), strcspn(), strlcat(),strlcpy(), strlen(), strncat(),strncmp(), strncpy(), strpbrk(),strrchr(), strspn(), strstr(),strxfrm(), tan(), tanh()	这些函数在本质上就是线程安全的。
longjmp(), setjmp()	虽然 setjmp() 和 longjmp() 在 __user_libspace 中保存数据，但它们均调用线程安全的 __alloca_* 函数。
remove(), rename(), time()	这些函数使用中断，以便与 ARM 调试环境进行通信。通常，必须为实际应用程序重新实现这些函数。
snprintf(), sprintf(), vsnprintf(),vsprintf(), sscanf(), isalnum(),isalpha(), iscntrl(), isdigit(),isgraph(), islower(), isprint(),ispunct(), isspace(), isupper(),isxdigit(), tolower(), toupper(),strcoll(), strtod(), strtol(),strtoul(), strftime()	使用这些函数时，这些基于字符串的函数将读取语言环境。通常，它们是线程安全的。但是，如果在会话中更改语言环境，则必须确保这些函数不受影响。 基于字符串的函数并不依赖于 stdio 库，例如， sprintf() 和 sscanf()。
stdin, stdout, stderr	这些函数是线程安全的。

FP 状态字

可以在多线程环境 （甚至软件浮点）中安全地使用 FP 状态字。其中，每个线程的状态字存储在其自己的 __user_perthread_libspace 块中。

注意

请注意，在硬件浮点中，FP 状态字存储在 VFP 寄存器中。在这种情况下，线程切换机制必须为每个线程保留该寄存器的单独副本。

非线程安全的函数

表 2-2 显示了非线程安全的 C 库函数。

表 2-2 非线程安全的函数

函数	说明
setlocale()	语言环境设置是所有线程的全局设置，并且未使用锁对其进行保护。如果两个线程调用 setlocale()，则可能会发生数据损坏。另外，很多其他函数读取当前语言环境设置，例如，strtod() 和 sprintf()。因此，如果一个线程调用 setlocale()，另一个线程同时调用此函数，则可能会产生意外结果。 ARM 建议您选择所需的语言环境，然后调用一次 setlocale() 以对其进行初始化。应在程序中创建任何其他线程之前执行此操作，以使任意数量的线程可以同时读取语言环境设置，而不会相互干扰。 请注意，localeconv() 不是线程安全的。应改用指向用户提供的缓冲区的指针调用 ARM 函数 _get_lconv()。
asctime(), localtime(), strtok()	这些函数不是线程安全的。每个函数都包含一个静态缓冲区，其他线程可能会在调用函数以及随后使用其返回值之间覆盖该缓冲区。 ARM 提供了可重入版本 _asctime_r()、_localtime_r() 和 _strtok_r()。ARM 建议您改用这些函数以确保安全。 **注意** 这些可重入版本使用一些附加参数。_asctime_r() 使用的附加参数是指向输出字符串要写入的缓冲区的指针。_localtime_r() 使用的附加参数是指向结果要写入的 struct tm 的指针。_strtok_r() 使用的附加参数也是一个指针，指向的是指向下一个标记的 **char** 指针。

表 2-2 非线程安全的函数 （续）

函数	说明
gamma()[a], lgamma()	这些扩展 mathlib 函数使用全局变量 _signgam，因此不是线程安全的。
mbrlen(), mbsrtowcs(), mbrtowc(),wcrtomb(), wcsrtombs()	stdlib.h 中定义的 C89 多字节转换函数 （如 mblen() 和 mbtowc()）不是线程安全的，因为它们包含在所有线程之间共享而没有锁定的内部静态状态。 但是， wchar.h 中定义的扩展可重启版本 （例如，mbrtowc() 和 wcrtomb()）是线程安全的，但前提是您传入指向您自己的 mbstate_t 对象的指针。如果要在处理多字节字符串时确保线程安全，这些函数只能使用非 NULL 的 mbstate_t * 参数。
exit()	即使提供了基本 _sys_exit() （实际终止所有线程）的实现，也不要在多线程程序中调用 exit()。 在这种情况下， exit() 在调用 _sys_exit() *之前* 先执行清除操作，因此会中断其他线程。
rand(), srand()	这些函数保留全局性且不受保护的内部状态。这意味着， rand() 调用从来都不是线程安全的。 ARM 建议您使用自己的锁定，以确保每次只有一个线程调用 rand()，例如，通过定义 $Sub$$rand() （如果要避免更改代码）。 或者，也可以执行以下操作之一： • 提供您自己的随机数生成器，它可能具有多个独立实例 • 硬性规定只有一个线程需要生成随机数。

a. 已不提倡使用 gamma()。

2.2.7 ARM C++ 库中的线程安全性

下面简要介绍了 C++ 库中的线程安全性：

- `std::set_new_handler()` 函数不是线程安全的。这意味着，就 `std::set_new_handler()` 而言，某些形式的 `::operator new` 和 `::operator delete` 不是线程安全的。
  - 以下各项的缺省 C++ 运行时库实现使用 `malloc()` 和 `free()`，并且彼此之间都是线程安全的。就 `std::set_new_handler()` 而言，它们不是线程安全的。允许对它们进行替换：

    ```
    ::operator new(std::size_t)
    ::operator new[](std::size_t)
    ::operator new(std::size_t, const std::nothrow_t&)
    ::operator new[](std::size_t, const std::nothrow_t)
    ::operator delete(void*)
    ::operator delete[](void*)
    ::operator delete(void*, const std::nothrow_t&)
    ::operator delete[](void*, const std::nothrow_t&)
    ```

  - 下面的配置形式也是线程安全的。不允许对它们进行替换：

    ```
    ::operator new(std::size_t, void*)
    ::operator new[](std::size_t, void*)
    ::operator delete(void*, void*)
    ::operator delete[](void*, void*)
    ```

- 全局对象的构建和析构不是线程安全的。
- 如果适当地重新实现 `__cxa_guard_acquire()`、`__cxa_guard_release()`、`__cxa_guard_abort()`、`__cxa_atexit()` 和 `__aeabi_atexit()` 函数，则可以将局部静态对象构建变为线程安全的操作。例如，通过进行适当的重新实现，可以将以下 `lsobj` 构建变为线程安全的操作：

  ```
  struct T { T(); };
  void f() { static T lsobj; }
  ```

  有关 `__cxa_` 和 `__aeabi_` 函数的信息，请参阅 CPPABI。
- 如果调用的任何用户构造函数和析构函数是线程安全的，则引发异常也是线程安全的。有关详细信息，请参阅 《ARM *体系结构的异常处理 ABI*》。
- ARM C++ 库使用 ARM C 库。要在多线程环境中使用 ARM C++ 库，您必须提供第2-10 页的*将 ARM C 库用于多线程应用程序*中所述的函数。

2.3 使用 C 库构建应用程序

本节介绍了如何创建与 C 或 C++ 库中的函数链接的应用程序。C 库中的函数负责：

- 创建一个可以在其中执行 C 或 C++ 程序的环境。这包括：
  - 创建一个堆栈
  - 创建一个堆 （如果需要）
  - 初始化程序所用的库的部分组成内容。
- 调用 `main()` 以开始执行程序。
- 支持程序使用 ISO 定义的函数。
- 捕获运行时错误和信号，如果需要，还可以在出现错误或程序退出时终止执行。

2.3.1 将库用于应用程序

可以使用三种方法将库用于应用程序：

- 构建一个可在半主机环境中调试的半主机应用程序，如具有 RealView ISS、ISSM、RealView ICE 或 RealMonitor 的环境。请参阅*构建用于半主机环境的应用程序*。
- 构建一个非主机应用程序，例如，可嵌入到 ROM 中的应用程序。请参阅第2-19 页的*构建用于非半主机环境的应用程序*。
- 构建一个不使用 `main()` 且不初始化库的应用程序。除非重新实现某些函数，否则，这种应用程序具有有限的库功能。请参阅第2-24 页的*不使用 C 库构建应用程序*。

2.3.2 构建用于半主机环境的应用程序

如果开发在半主机环境中运行以进行调试的应用程序，您必须有一个支持 ARM 或 Thumb 半主机的执行环境，并且具有足够的内存。

可以使用以下任一方法来提供执行环境：

- 使用 RealView ISS、 ISSM、 RealView ICE 和 RealMonitor 等在缺省情况下提供的标准半主机功能
- 实现您自己的半主机调用处理程序。请参阅 《开发指南》中的第 8 章 *半主机*。

有关需要半主机的函数的列表，请参阅第2-20 页的*半主机相关性概述*。

如果使用库的缺省半主机功能，则不需要编写任何新的函数或包含文件。

使用 RealView ISS 或 ISSM

RealView ISS 和 ISSM 支持半主机，并且具有允许使用库的内存映射。RealView ISS 和 ISSM 使用主机中的内存，对于应用程序来说，这通常就足够了。

使用 RealView ICE

ARM 调试代理支持半主机，但可能需要调整库所使用的内存映射，以便与所调试的硬件相匹配。不过，调整 C 库使用的内存映射比较容易。请参阅第2-67 页的*调整运行时内存模型*。

在半主机环境中使用重新实现的函数

也可以将半主机功能与新的输入/输出函数混合使用。例如，除了半主机实现以外，还可以实现 `fputc()` 以直接输出到硬件，如 UART。有关如何重新实现各个函数的信息，请参阅*构建用于非半主机环境的应用程序*。

将半主机应用程序转换为独立应用程序

在半主机调试环境中开发应用程序后，可以使用以下方法之一将其移到非主机环境中：

- 删除对半主机函数的所有调用。请参阅第2-22 页的*避免使用半主机*。
- 重新实现低级函数，例如 `fputc()`。请参阅*构建用于非半主机环境的应用程序*。不必重新实现所有半主机函数。但是，您必须重新实现在应用程序中使用的函数。
- 实现所有半主机调用的处理程序。

2.3.3 构建用于非半主机环境的应用程序

如果不想使用任何半主机功能，则必须删除对半主机函数的所有调用，或者使用非半主机函数重新实现它们。

要构建不使用半主机功能的应用程序，请执行以下操作：

1. 创建源文件以实现与目标相关的功能。例如，使用半主机调用或依赖于目标内存映射的函数。

2. 将 __use_no_semihosting 符号添加到源文件中。请参阅第2-22 页的*避免使用半主机*。
3. 将新对象与应用程序进行链接。
4. 在创建与目标相关的应用程序时使用新配置。

必须重新实现 C 库所使用的函数，以使其自身不再具有目标相关性。例如，如果使用 fputc() 函数，则必须重新实现 printf() 函数。如果不使用高级输入/输出函数 （如 printf()），则不必重新实现低级函数 （如 fputc()）。

如果为不同的执行环境构建应用程序，则可以重新实现与目标相关的函数。例如，使用半主机调用或依赖于目标内存映射的函数。C++ 库中没有与目标相关的函数，但某些 C++ 函数使用与目标相关的基本 C 库函数。

主示例目录的 ...\emb_sw_dev 中包含不使用主机环境的嵌入式应用程序示例。

有关创建嵌入式应用程序的示例，请参阅 《开发指南》。

非半主机环境中的 C++ 异常

C++ 标准需要使用缺省 C++ abort() 处理程序来调用 std::terminate()。abort() 的缺省 C 库实现使用需要半主机支持的函数。因此，如果在非半主机环境中使用异常，则必须提供 abort() 的替代实现。

半主机相关性概述

表 2-3 显示了直接依赖于半主机的函数。

表 2-3 直接半主机相关性

函数	说明
__user_initial_stackheap()	请参阅第2-67 页的*调整运行时内存模型*。如果使用分散加载，则可能需要重新实现此函数。

表 2-3 直接半主机相关性 （续）

函数	说明
_sys_exit() _ttywrch()	请参阅第2-58 页的*调整错误信号、错误处理和程序退出*。
_sys_command_string(), _sys_close(), _sys_ensure(), _sys_iserror(), _sys_istty(), _sys_flen(), _sys_open(), _sys_read(), _sys_seek(), _sys_write(), _sys_tmpnam()	请参阅第2-76 页的*调整输入/输出函数*。
clock(), _clock_init(), remove(), rename(), system(), time()	请参阅第2-90 页的*调整其他 C 库函数*。

表 2-4 显示的函数间接依赖于第2-20 页的表 2-3 中列出的一个或多个函数。

表 2-4 间接半主机相关性

函数	用法
__raise()	捕获、处理或诊断 C 库异常，没有 C 信号支持。请参阅第2-58 页的*调整错误信号、错误处理和程序退出*。
__default_signal_handler()	捕获、处理或诊断 C 库异常，具有 C 信号支持。请参阅第2-58 页的*调整错误信号、错误处理和程序退出*。
__Heap_Initialize()	选择或重新定义内存分配。请参阅第2-64 页的*调整存储管理*。
ferror(), fputc(), __stdout	重新实现 printf 系列。请参阅第2-76 页的*调整输入/输出函数*。
__backspace(), fgetc(), __stdin	重新实现 scanf 系列。请参阅第2-76 页的*调整输入/输出函数*。
fwrite(), fputs(), puts(),fread(), fgets(), gets(), ferror()	重新实现流输出系列。请参阅第2-76 页的*调整输入/输出函数*。

避免使用半主机

如果使用 C 编写应用程序，则必须将其与 C 库进行链接，即使应用程序不直接使用 C 库函数。C 库包含编译器辅助函数和初始化代码。某些 C 库函数使用半主机。

要避免使用半主机，请执行以下任一操作：

- 在您自己的应用程序中重新实现这些函数
- 按某种方式编写应用程序，以使其不调用任何半主机函数。

要确保应用程序中不包含任何使用半主机的函数，请使用以下任一方法：

- 汇编语言中的 `IMPORT __use_no_semihosting`
- C 中的 `#pragma import(__use_no_semihosting)`。

注意

`IMPORT __use_no_semihosting` 只需添加到一个汇编源文件中。同样，`#pragma import(__use_no_semihosting)` 只需添加到一个 C 源文件中。不需要将这些插入添加到每个源文件中。

如果包含使用半主机的库函数，并且还引用了 `__use_no_semihosting`，库将检测到符号冲突，并且链接器将报告错误。要找出使用半主机的对象，请使用 `--verbose --list=out.txt` 进行链接，搜索输出以查找符号，然后找出引用该符号的对象。有关详细信息，请参阅 《链接器参考指南》中第2-35 页的*--list=file* 和第2-59 页的*--verbose*。

API 定义

除了第2-20 页的表 2-3 和第2-21 页的表 2-4 中列出的半主机函数外，表 2-5 还显示了为不同环境构建应用程序时可能用到的函数和文件。

表 2-5 已公布的 API 定义

文件或函数	说明
`__main()` `__rt_entry()`	初始化运行时环境并执行用户应用程序。
`__rt_lib_init()`, `__rt_exit()`, `__rt_lib_shutdown()`	初始化或结束运行时库。
`LC_CTYPE locale`	为局部字母表定义字符属性。请参阅第2-41 页的*使用汇编器宏调整语言环境和 CTYPE*。

表 2-5 已公布的 API 定义 （续）

文件或函数	说明
rt_sys.h	C 头文件，用于描述缺省 （半主机）实现使用半主机调用的所有函数。
rt_heap.h	C 头文件，用于描述存储管理抽象数据类型。
rt_locale.h	C 头文件，用于描述 5 个语言环境类别*文件编排系统*，并定义一些用于描述语言环境类别内容的宏。
rt_misc.h	C 头文件，用于描述 C 库的其他无关公共接口。
rt_memory.s	内存模型的带注释空原型实现。有关该文件的说明，请参阅第2-68 页的*编写您自己的内存模型*。

如果重新实现标准 ARM 库中存在的函数，链接器将使用项目中的对象或库，而不是使用标准 ARM 库。添加到项目中的任何库不必遵循 ARM 库命名约定。

小心

不要替换或删除 ARM Limited 提供的库。切勿覆盖提供的库文件，而应将重新实现的函数放在单独的对象文件或库中。

2.4 不使用 C 库构建应用程序

如果创建的应用程序包含 main() 函数，则会导致将 C 库初始化函数包含在 __rt_lib_init 中。

如果应用程序不包含 main() 函数，则不会初始化 C 库，并且不能在应用程序中使用以下功能：

- 带有前缀 _sys_ 的低级 stdio 函数
- signal.h 中的信号处理函数 signal() 和 raise()
- 其他函数，如 atexit() 和 alloca()。

有关不进行库初始化就无法使用的函数的详细信息，请参阅第2-27 页的*独立 C 库函数*。

本节介绍如何创建不包含该库的应用程序 （作为 *裸机 C*）。这些应用程序不会自动使用 C 库提供的完整 C 运行时环境。即使创建的应用程序不包含该库，也必须包含该库中的某些辅助函数。还可以通过少量的重新实现将很多库函数变为可用。

2.4.1 整数和浮点辅助函数

编译器使用一些编译器辅助函数来处理没有等效短机器代码的运算。例如，如果目标指令集中未提供除法指令，则整数除法需要使用辅助函数。（ARMv7-R 和 ARMv7-M 体系结构使用 Thumb 状态的指令 SDIV 和 UDIV。其他版本的 ARM 体系结构使用编译器辅助函数。）

整数除法和所有浮点函数需要使用 __rt_raise() 来处理数学错误。重新实现 __rt_raise() 可启用所有数学辅助函数，并且不再需要在所有信号处理库代码中进行链接。

2.4.2 裸机整数 C

如果使用 C 编写的程序不使用库，并且无需进行任何环境初始化即可运行，则必须：

- 重新实现 __rt_raise()，因为可能会从编译的代码中的多个位置调用该错误处理函数。
- 不能定义 main() 以避免在库初始化代码中进行链接。
- 编写汇编语言中间代码，它用于确定运行 C 所需的寄存器状态。该中间代码必须跳转到应用程序的入口函数。

- 提供您自己的 RW/ZI 初始化代码。
- 确保执行初始化中间代码，例如，通过将其放在复位处理程序中。
- 使用 --fpu=none 构建应用程序并正常进行链接。链接器使用相应的 C 库变体查找任何所需的编译器辅助函数。

很多库功能需要使用 __user_libspace 来处理静态数据。即使没有使用 main() 函数激活初始化代码，也会自动创建 __user_libspace，并且它使用 ZI 段中的 96 个字节。有关 __user_libspace 区的说明，请参阅第2-6 页的*__user_libspace 静态数据区*。

2.4.3 具有浮点功能的裸机 C

如果要在应用程序中使用浮点处理，您必须：

- 执行整数 C 所需的步骤，如第2-24 页的*裸机整数 C*中所述。但是，不要使用 --fpu=none 选项构建应用程序。
- 在构建应用程序时使用相应的 FPU 选项。
- 在执行任何浮点运算之前，调用 _fp_init() 以初始化浮点状态寄存器。

如果使用的是软件浮点，还可以定义 __rt_fp_status_addr() 函数以返回要使用的可写数据字的地址，而不是浮点状态寄存器。如果不这样做，则会创建占用 96 个字节的 __user_libspace 区。有关 __user_libspace 区的说明，请参阅第2-6 页的*__user_libspace 静态数据区*。

2.4.4 使用 C 库

如果创建的应用程序包含 main() 函数，链接器将自动包含执行环境所需的初始化代码。有关说明，请参阅第2-18 页的*使用 C 库构建应用程序*。但在某些情况下，您可能不希望或无法这样做。

可以创建一个由自定义启动代码组成的应用程序，并且应用程序仍使用很多库函数。您必须满足以下条件之一：

- 避免使用需要初始化的函数
- 提供初始化和低级支持函数。

程序设计

必须重新实现的函数取决于您需要多少库功能：

- 如果仅需要用于除法、结构复制和 FP 算法的编译器支持函数，则必须提供 __rt_raise()。这还可以启用非常简单的库函数，如 errno.h 和 setjmp.h 中的函数以及 string.h 中的大多数函数。
- 如果显式地调用 setlocale()，则会激活与语言环境相关的函数。这样，您便可使用 atoi 系列、 sprintf()、 sscanf() 以及 ctype.h 中的函数。
- 使用浮点的程序必须调用 _fp_init()。如果选择软件浮点，程序还必须提供 __rt_fp_status_addr()。如果未重新实现此函数，则缺省操作是创建一个 __user_libspace 区。有关 __user_libspace 区的说明，请参阅第2-6 页的 *__user_libspace 静态数据区*。
- 对于使用 fprintf() 或 fputs() 的函数，必须实现高级输入/输出支持。高级输出函数依赖于 fputc() 和 ferror()。高级输入函数依赖于 fgetc() 和 __backspace()。

通过实现这些函数和堆，您几乎可以使用整个库。

使用低级函数

如果在没有 main() 函数的应用程序中使用库，则必须重新实现库中的某些函数。有关详细信息，请参阅第2-27 页的*独立 C 库函数*。

__rt_raise() 是基本函数。所有 FP 函数、整数除法 （以便在除数为零时报告出现错误）以及某些其他库例程都需要使用此函数。如果不实现一些需要 __rt_raise() 的操作，则可能无法编写出好的程序。

注意

如果要调用 rand()，则*必须* 先调用 srand()。这是在库初始化期间自动完成的，但如果避免进行库初始化，则不会自动完成。

使用高级函数

如果重新实现了低级函数 （例如 fputc()），则可以使用高级 I/O 函数 （例如 fprintf()）。大多数设置了格式的输出函数也需要调用 setlocale()。有关说明，请参阅第2-76 页的*调整输入/输出函数*。

要调用使用 locale 的任何内容，必须先调用 setlocale() 以对其进行初始化，例如，调用 setlocale(LC_ALL, "C")。*独立 C 库函数*中介绍了使用语言环境的函数，其中包括 ctype.h 和 locale.h 中的函数、printf() 系列、scanf() 系列、ato*、strto*、strcoll/strxfrm 以及 time.h 中的大多数函数。

使用 malloc()

如果裸机 C 需要堆支持，则必须先调用 _init_alloc() 以提供初始堆范围，并且*必须*提供 __rt_heap_extend()，即使它只返回错误。rt_heap.h 中提供了这两个函数的原型。

2.4.5 独立 C 库函数

本节的其余部分中列出了未初始化的库中提供的包含文件及其包含的函数。对于某些无法使用的函数，可通过重新实现它们所依赖的库函数将其变为可用。

alloca.h

如果未进行库初始化，则无法使用此文件中列出的函数。有关说明，请参阅第2-18 页的*使用 C 库构建应用程序*。

assert.h

此文件中列出的函数需要高级 stdio、__rt_raise() 和 _sys_exit()。有关说明，请参阅第2-58 页的*调整错误信号、错误处理和程序退出*。

ctype.h

此文件中列出的函数需要 locale 函数。

errno.h

此文件中的函数不需要进行任何库初始化或函数重新实现即可使用。

fenv.h

此文件中的函数不需要进行任何库初始化，而只需重新实现 __rt_raise() 即可使用。

float.h

此文件不包含任何代码。该文件中的定义不需要进行库初始化或函数重新实现。

inttypes.h

此文件中列出的函数需要 locale 函数。

limits.h

此文件中的函数不需要进行任何库初始化或函数重新实现即可使用。

locale.h

在调用使用 locale 函数的任何函数之前，应先调用 setlocale()。例如：

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

请参阅 locale.h 的内容，以了解下列函数和数据结构的详细信息：

- setlocale() 选择 category 和 locale 参数指定的相应语言环境。
- lconv 是由 locale 函数使用的一种结构，用于依照当前语言环境规则设置数量的格式。
- localeconv() 创建一个 lconv 结构，并返回指向该结构的指针。
- _get_lconv() 填充参数所指向的 lconv 结构。此 ISO 扩展删除了对库中静态数据的要求。

locale.h 还包含与 locale 函数一起使用的常数声明。有关详细信息，请参阅第2-41 页的 *使用汇编器宏调整语言环境和 CTYPE*。

math.h

要能够使用此文件中的函数，必须先调用 _fp_init() 并重新实现 __rt_raise()。

setjmp.h

此文件中的函数不需要进行任何库初始化或函数重新实现即可使用。

signal.h

如果未进行库初始化，则无法使用此文件中列出的函数。有关构建使用库初始化的应用程序的说明，请参阅第2-18 页的*使用 C 库构建应用程序*。

可以重新实现 __rt_raise() 以进行错误和退出处理。有关说明，请参阅第2-58 页的*调整错误信号、错误处理和程序退出*。

stdarg.h

此文件中列出的函数不需要进行任何库初始化或函数重新实现即可使用。

stddef.h

此文件不包含任何代码。该文件中的定义不需要进行库初始化或函数重新实现。

stdint.h

此文件不包含任何代码。该文件中的定义不需要进行库初始化或函数重新实现。

stdio.h

以下相关性或限制适用于这些函数：

- 高级函数 （如 printf()、 scanf()、 puts()、 fgets()、 fread()、 fwrite() 和 perror()）依赖于低级 stdio 函数 fgetc()、 fputc() 和 __backspace()。使用独立 C 库时，必须重新实现这些低级函数。

  但是，在使用独立 C 库时，无法重新实现带有 _sys_ 前缀的函数 （例如 _sys_read()），因为它们需要进行库初始化。

  有关详细信息，请参阅第2-76 页的*调整输入/输出函数*。

- printf() 和 scanf() 函数系列需要 locale。

- remove() 和 rename() 函数是系统特有的函数，可能无法在您的应用程序中使用。

stdlib.h

此文件中的大多数函数不需要进行任何库初始化或函数重新实现即可使用。以下函数依赖于正确实例化的其他函数：

- ato*() 需要 locale
- strto*() 需要 locale
- malloc()、 calloc()、 realloc() 和 free() 需要堆函数
- 不使用 C 库构建应用程序时，将无法使用 atexit()。

string.h

此文件中的函数不需要进行任何库初始化即可使用，但 strcoll() 和 strxfrm() 例外，它们需要 locale。

time.h

- mktime() 和 localtime() 可直接使用
- time() 和 clock() 是系统特定的函数，不经重新实现可能无法使用。
- asctime()、 ctime() 和 strftime() 需要 locale。

wchar.h

1994 年，*标准附录 1* 将宽字符库函数添加到 ISO C 中。

- 支持宽字符输出和格式字符串：swprintf()、 vswprintf()、 swscanf() 和 vswscanf()
- 所有转换函数 （如 btowc、 wctob、 mbrtowc 和 wcrtomb）都需要 locale
- wcscoll 和 wcsxfrm 需要 locale。

wctype.h

1994 年，*标准附录 1* 将宽字符库函数添加到 ISO C 中。这需要 locale。

2.5 调整 C 库以适应新的执行环境

本节介绍如何重新实现函数以开发用于不同执行环境的应用程序，例如，嵌入到 ROM 中或与 RTOS 一起使用的应用程序。

以单下划线或双下划线开头的符号用于命名用作低级实现的组成部分的函数。您可以重新实现其中的某些函数。

rt_heap.h、rt_locale.h、rt_misc.h 和 rt_sys.h 包含文件以及 rt_memory.s 汇编器文件中提供了有关这些库函数的附加说明。

有关带有 __cxa 或 __aeabi 前缀的函数的说明，请参阅以下规范：

- *《ARM 体系结构的 C 库 ABI》*
- *《ARM 体系结构的异常处理 ABI》*
- *《ARM 体系结构的 C++ ABI》*

2.5.1 C 和 C++ 程序使用库函数的方式

本节介绍以下内容：

- 用于初始化执行环境和应用程序的特定库函数
- 库退出函数
- 应用程序本身在执行期间可能会调用的目标相关库函数

初始化执行环境并执行应用程序

程序入口点位于 C 库中的 __main 处，库代码在此处执行以下操作：

1. 将非根 （RO 和 RW）执行区从其加载地址复制到执行地址。另外，如果压缩了任何数据节，则会将它们从加载地址解压缩到执行地址。有关详细信息，请参阅 《链接器参考指南》。
2. 将 ZI 区清零。
3. 跳转到 __rt_entry。

如果不希望库执行这些操作，您可以定义自己的 __main 以跳转到 __rt_entry，如第2-32 页的示例 2-1 所示。

示例 2-1 __main 和 __rt_entry

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

库函数 __rt_entry() 按以下方式运行程序：

1. 调用 __rt_stackheap_init() 以设置堆栈和堆。
2. 调用 __rt_lib_init() 以初始化引用的库函数、初始化语言环境以及为 main() 设置 argc 和 argv （如果需要）。

   对于 C++，通过 __cpp_initialize__aeabi_ 为任何顶层对象调用构造函数。有关详细信息，请参阅*C++ 初始化、构建和析构*。
3. 调用 main() （应用程序的用户级根）。

   从 main() 中，程序可以调用库函数等一些函数。有关详细信息，请参阅第2-35 页的*从 main() 中调用的库函数*。
4. 使用 main() 返回的值调用 exit()。

C++ 初始化、构建和析构

C++ 对构建和析构具有静态存储时限的对象设定了一些要求。请参阅 *C++ 标准的第 3.6 节*。

ARM C++ 编译器使用 .init_array 区来实现此目的。这是一个指向函数的自相关指针的常数数据数组。例如，可以在 test.cpp 文件中包含以下 C++ 转换单元：

```
struct T
{
    T();
    ~T();
} t;
int f()
{
    return 4;
}
int i = f();
```

这会转换为以下伪代码：

```
        AREA ||.text||, CODE, READONLY
     int f()
    {
        return 4;
    }
     static void __sti___8_test_cpp
     {
        // construct 't' and register its destruction
        __aeabi_atexit(T::T(&t), &T::~T, &__dso_handle);
        i = f();
     }
        AREA ||.init_array||, DATA, READONLY
        DCD __sti___8_test_cpp - {PC}
        AREA ||.data||, DATA
     t   % 4
     i   % 4
```

此伪代码仅用于说明目的。要查看生成的代码，请使用 armcc -c --cpp -S 编译 C++ 源代码。

链接器将不同转换单元中的每个 .init_array 收集在一起。按相同顺序累积 .init_array 是非常重要的。

库例程 __cpp_initialize__aeabi_ 是在 main 之前从 C 库启动代码 __rt_lib_init 中调用的。__cpp_initialize__aeabi_ 遍历 .init_array 以依次调用每个函数。在出口处，__rt_lib_shutdown 调用 __cxa_finalize。

通常，每个转换单元最多有一个用于 T::T() 的函数 （重整名称为 _ZN1TC1Ev）、一个用于 T::~T() 的函数 （重整名称为 _ZN1TD1Ev）、一个 __sti__ 函数以及 4 个字节的 .init_array。f() 函数的重整名称为 _Z1fv。无法确定转换单元之间的初始化顺序。

带有析构函数的函数局部静态对象也是使用 __aeabi_atexit 处理的。

.init_array 节须连续放置在同一区中，以便能够访问其 base/limit 符号。否则，链接器将产生错误。

旧对象支持

RVCT v2.0 及更低版本中使用 C$$pi_ctorvec 代替了 .init_array。具有 C$$pi_ctorvec 的对象仍然受到支持。因此，如果有旧对象，分散文件应包含：

```
LOAD_ROM 0x0000000
{
    EXEC_ROM 0x0000000
    {
        your_object.o(+RO)
```

```
        * (.init_array)
        * (C$$pi_ctorvec)        ; backwards compatibility
        ...
    }
    RAM 0x0100000
    {
        * (+RW,+ZI)
    }
}
```

异常系统初始化

可以在需要时 （即在第一次使用时）初始化异常系统，也可以在进入 main 之前对其进行初始化。在需要时进行初始化的优点是，除非使用异常系统，否则不分配堆内存；但缺点是如果在第一次使用时堆耗尽，则无法引发任何异常 （如 std::bad_alloc）。

缺省设置是在需要时进行初始化。要在进入 main 之前初始化异常系统，请在链接中包含以下函数：

```
extern "C" void __cxa_get_globals(void);
extern "C" void __ARM_exceptions_init(void)
{
    __cxa_get_globals();
}
```

虽然可以将 __cxa_get_globals 调用直接放在代码中，但将它放在 __ARM_exceptions_init 中可确保尽早对其进行调用。即，在初始化任何全局变量和进入 main 之前。

库初始化机制弱引用 __ARM_exceptions_init，并在将其作为 __rt_lib_init 的一部分时调用它。

注意

异常系统是通过调用不同库函数来进行初始化的，例如，std::set_terminate()。因此，在进入 main 之前，可能不需要进行初始化。

用于异常的紧急缓冲区内存

可以选择是否分配紧急内存，它用于在堆耗尽时引发 std::bad_alloc 异常。

要分配紧急内存，您必须在链接中包含符号 __ARM_exceptions_buffer_required。随后，将调用 __ARM_exceptions_buffer_init() 并将其作为异常系统初始化的一部分。缺省情况下，不包含该符号。

以下例程用于管理异常紧急缓冲区：

`extern "C" void *__ARM_exceptions_buffer_init()`

在运行时调用一次，以分配紧急缓冲区内存。它返回指向紧急缓冲区内存的指针；如果未分配内存，则返回 NULL。

`extern "C" void *__ARM_exceptions_buffer_allocate(void *buffer, size_t size)`

在将要引发异常但没有足够可用堆内存来分配异常对象时调用。*buffer* 是 __ARM_exceptions_buffer_init() 以前返回的值；如果没有调用该例程，则返回 NULL。__ARM_exceptions_buffer_allocate() 返回指向在 8 字节边界上对齐的 *size* 个字节内存的指针；如果无法分配，则返回 NULL。

`extern "C" void *__ARM_exceptions_buffer_free(void *buffer, void *addr)`

调用以释放可能由 __ARM_exceptions_buffer_allocate() 分配的内存。*buffer* 是 __ARM_exceptions_buffer_init() 以前返回的值；如果未调用该例程，则返回 NULL。该例程确定是否已从紧急内存缓冲区中分配了传递的地址；如果已分配，则会适当地将其释放，然后返回非 NULL 值。如果 __ARM_exceptions_buffer_allocate() 未分配位于 *addr* 的内存，该例程必须返回 NULL。

这些例程的缺省定义存在于映像中，但您可以提供自己的版本以覆盖库提供的缺省定义。缺省例程只保留足够用于单个 std::bad_alloc 异常对象的空间。如果不需要紧急缓冲区，则可以安全地将所有这些例程重新定义为只返回 NULL。

从 main() 中调用的库函数

main() 函数是应用程序的用户级根。它要求初始化执行环境，并且要求能够调用输入/输出函数。在 main() 中，程序可以执行以下操作之一（用于在 C 库中调用用户可自定义的函数）：

- 扩展堆栈或堆。请参阅第 2-67 页的 *调整运行时内存模型*。
- 调用一些需要调用用户定义的函数的库函数，例如，__rt_fp_status_addr() 或 clock()。请参阅第 2-90 页的 *调整其他 C 库函数*。
- 调用一些使用 locale 或 CTYPE 的库函数。请参阅第 2-41 页的 *使用汇编器宏调整语言环境和 CTYPE*。
- 执行需要 fpu 或浮点库的浮点计算。

- 通过低级函数 （如 putc()）直接输入或输出；或者通过高级输入/输出函数和输入/输出支持函数 （如 fprintf() 或 sys_open()）间接输入或输出。请参阅第2-76 页的*调整输入/输出函数*。
- 产生错误或其他信号，例如， ferror。请参阅第2-58 页的*调整错误信号、错误处理和程序退出*。

2.5.2 __rt_entry

__rt_entry 符号是使用 ARM C 库的程序的起点。将所有分散加载区重定位到其执行地址后，会将控制权传递给 __rt_entry。

用法

__rt_entry 的缺省实现：

1. 设置堆和堆栈。
2. 调用 __rt_lib_init 以初始化 C 库。
3. 调用 main()。
4. 调用 __rt_lib_shutdown 以关闭 C 库。
5. 退出。

__rt_entry 必须以调用以下函数之一结束：

exit() 调用 atexit() 注册的函数并关闭库。

__rt_exit() 关闭库，但不调用 atexit() 函数。

_sys_exit() 直接退出到执行环境。它不关闭库，也不调用 atexit() 函数。请参阅第2-59 页的*_sys_exit()*。

2.5.3 从程序中退出

程序可以在 main() 末尾正常退出，或者在出现错误时提前退出。

从声明中退出

assert 宏的行为取决于最近出现的 #include <assert.h> 的运行条件。

1. 如果定义了 NDEBUG 宏 （在命令行中或作为源文件的一部分），则 assert 宏无效。

2. 如果没有定义 NDEBUG 宏，则对 assert 表达式（为 assert 宏指定的表达式）进行求值。如果结果是 TRUE （即 != 0），则 assert 宏不再有效。

3. 如果 assert 表达式的计算结果为 FALSE，并且任何以下条件成立， assert 宏将调用 __aeabi_assert() 函数。
   - 使用 --strict 进行编译
   - 使用 -O0 或 -O1
   - 使用 --library_interface=aeabi_clib 或 --library_interface=aeabi_glibc 进行编译
   - 定义了 __ASSERT_MSG
   - 定义了 _AEABI_PORTABILITY_LEVEL 并且它不为 0。

4. 否则，如果 assert 表达式的计算结果为 FALSE，并且上面第 3 点中指定的条件不适用， assert 宏将调用 abort()。然后：
   a. abort() 调用 __rt_raise()。
   b. 如果 __rt_raise() 返回结果， abort() 将尝试结束库。

如果创建的应用程序不使用库，并且重新实现了 abort() 和 stdio 函数，则 __aeabi_assert() 有效。

另一个目标重定向解决方案是重新实现 __aeabi_assert() 函数本身。函数原型是：

```
void __aeabi_assert(const char *expr, const char *file, int line);
```

其中：
- *expr* 指向非 TRUE 表达式的字符串表示形式
- *file* 和 *line* 指定断言的源位置。

ARM C 库中提供的 __aeabi_assert() 的行为是在 stderr 中输出消息并调用 abort()。

可通过在较高优化级别定义 __ASSERT_MSG 来恢复 __aeabi_assert() 的缺省行为。

2.5.4 __rt_exit()

此函数关闭库，但不调用使用 atexit() 注册的函数。

语法

```
void __rt_exit(int code)
```

其中，标准函数不使用 *code*。

用法

通过调用 __rt_lib_shutdown 关闭 C 库，然后调用 _sys_exit 以终止应用程序。重新实现 _sys_exit，而不是 __rt_exit。有关详细信息，请参阅第2-59 页的 *_sys_exit()*。

返回值

此函数不返回值。

2.5.5 __rt_lib_init()

这是库初始化函数，它与 __rt_lib_shutdown() 配合使用。

语法

```
extern value_in_regs struct __argc_argv __rt_lib_init(unsigned heapbase,
unsigned heaptop)
```

其中：

heapbase 堆内存块起点。

heaptop 堆内存块终点。

用法

这是库初始化函数。它是紧靠 __rt_stackheap_init() 后面调用的，并传递一个要用作堆的初始内存块。此函数是标准 ARM 库初始化函数，不能重新实现此函数。

返回值

此函数返回 argc 和 argv，可随时将其传递给 main()。在寄存器中返回的结构是：

```
struct __argc_argv
{   int argc;
  char **argv;
  int r2, r3;
};
```

2.5.6 __rt_lib_shutdown()

这是库关闭函数，它与 __rt_lib_init() 配合使用。

语法

```
void __rt_lib_shutdown(void)
```

用法

这是库关闭函数，它是在用户需要直接对其进行调用时提供的。它是标准 ARM 库关闭函数，不能重新实现此函数。

2.6 调整静态数据访问

本节介绍使用 C 库中的调用来访问静态数据。可以将使用静态数据的 C 库函数划分为以下类别：

- 不使用任何类型静态数据的函数，例如， `fprintf()`
- 管理静态状态的函数，例如， `malloc()`、 `rand()` 和 `strtok()`
- 不管理静态状态但在 ARM 编译器中以实现特有的方式使用静态数据的函数，例如， `isalpha()`。

如果 C 库执行的某些操作需要隐式静态数据，则会使用对可替换函数的调用。表 2-6 中显示了这些函数。它们不使用半主机。

表 2-6 C 库调用

函数	说明
`__rt_errno_addr()`	调用以获取 `errno` 变量的地址。请参阅第2-60 页的 *__rt_errno_addr()*。
`__rt_fp_status_addr()`	浮点支持代码对其进行调用以获取浮点状态字的地址。请参阅第2-62 页的*__rt_fp_status_addr()*。
`locale` 函数	`__user_libspace()` 函数为库创建一个专用静态数据块。请参阅第2-41 页的*使用汇编器宏调整语言环境和 CTYPE*和第2-5 页的*编写可重入且线程安全的代码*。

有关内存使用的详细信息，另请参阅第2-67 页的*调整运行时内存模型*。

`__user_libspace` 的缺省实现在 ZI 段中创建一个 96 个字节的块。即使应用程序没有 `main()` 函数，通常也不需要重新定义 `__user_libspace()` 函数。但是，如果编写的是操作系统或进程切换程序，则必须重新实现此函数 （请参阅第2-5 页的*编写可重入且线程安全的代码*）。

——注意——

在将来的版本中，在定义中使用静态数据的具体函数可能会发生变化。

2.7 使用汇编器宏调整语言环境和 CTYPE

本节介绍了使用汇编器宏来调整语言环境函数。在显示或处理与本地语言或区域相关的数据时，应用程序将使用语言环境，例如，字符集、货币符号、小数点、时间和日期等。

有关与语言环境相关的函数的详细信息，请参阅 rt_locale.s 包含文件。

2.7.1 在链接时选择语言环境

可以在链接时选择 C 库的语言环境子系统，或者将其扩展为可以在运行时进行选择。下面介绍了库如何使用语言环境类别：

- 每个语言环境类别的缺省实现都以 C 语言环境为准。库还提供了另一种选择，您可以在链接时选择每个语言环境类别的 ISO8859-1 （Latin-1 字母表）实现。
- 通常， C 和 ISO8859-1 缺省实现为每个类别提供一个可在运行时选择的语言环境。
- 您可以分别替换每个语言环境类别。
- 您可以在每个类别中包含所选择的所有语言环境，并且可以命名选择的语言环境。
- 每个语言环境类别使用库专用静态数据中的一个字。
- 语言环境类别数据是只读的，并且与位置无关。
- scanf() 强制包含 LC_CTYPE 语言环境类别，但对于任一缺省语言环境，它仅在几千字节的代码中添加 260 个字节的只读数据。

ISO8859-1 实现

表 2-7 显示了 ISO8859-1 （Latin-1 字母表）语言环境类别。

表 2-7 缺省 ISO8859-1 语言环境

符号	说明
__use_iso8859_ctype	选择 ISO8859-1 (Latin-1) 字符分类。这实际上是 7 位 ASCII，只是字符代码 160-255 表示一组非常有用的欧洲标点符号字符、字母和重音字母。
__use_iso8859_collate	选择与 Latin-1 字母表对应的 strcoll/strxfrm 归类表。缺省 C 语言环境不需要归类表。
__use_iso8859_monetary	选择使用 Latin-1 编码的英国货币类别。
__use_iso8859_numeric	在输出数值时选择逗号作为千分位。
__use_iso8859_locale	选择此表中所述的所有 ISO8859-1 选项。

没有 ISO8859-1 版本的 LC_TIME 类别。

Shift-JIS 和 UTF-8 实现

表 2-8 显示了 Shift-JIS （日文字符）或 UTF-8 （Unicode 字符）语言环境类别。

表 2-8 缺省 Shift-JIS 和 UTF-8 语言环境

函数	说明
__use_sjis_ctype	将字符集设置为日文字符的 Shift-JIS 多字节编码
__use_utf8_ctype	将字符集设置为所有 Unicode 字符的 UTF-8 多字节编码

下面介绍了 Shift-JIS 编码的效果：

- 对于在 Shift-JIS 中为自包含字符的任何字节值，普通 ctype 函数可以正确对其进行处理。例如，isalpha() 将半宽度的片假名字符视为字母，Shift-JIS 将其编码为介于 0xA6 和 0xDF 之间的单个字节。
- 多字节转换函数（如 mbrtowc()、mbsrtowcs() 和 wcrtomb()）均可在 Unicode 宽字符串与 Shift-JIS 多字节字符串之间进行转换。

- printf("%ls") 将 Unicode 宽字符串转换为 Shift-JIS 输出；而 scanf("%ls") 将 Shift-JIS 输入转换为 Unicode 宽字符串。

如果应用程序中包含多个语言环境，则可以在运行时在多字节语言环境与单字节语言环境之间任意切换。缺省情况下，每次只包含一种语言环境。

2.7.2 在运行时选择语言环境

C 库函数 setlocale() 在运行时为其参数中指定的一个或多个语言环境类别选择语言环境。可通过在每个语言环境类别中分别选择所请求的语言环境来实现此目的。实际上，每个语言环境类别都是一个包含每种语言环境入口的小文件编排系统。

rt_locale.h 和 rt_locale.s 中给出了一些 C 头文件，它们描述了必须实现的内容，并提供了一些非常有用的支持宏。

2.7.3 定义语言环境块

语言环境数据块是使用 rt_locale.s 中提供的一组汇编语言宏定义的。因此，建议使用的语言环境块定义方法是编写汇编语言源文件。RVCT 为每种类型的语言环境数据块提供了一组宏，例如，LC_CTYPE、LC_COLLATE、LC_MONETARY、LC_NUMERIC 和 LC_TIME。可以使用 _begin 宏、一些数据宏以及 _end 宏，按相同方式定义每个语言环境块。

指定开头

要开始定义语言环境块，请调用 _begin 宏。该宏使用两个参数：前缀和文本名称。例如：

LC_*TYPE*_begin *prefix*, *name*

其中：

TYPE 是以下项之一：
- CTYPE
- COLLATE
- MONETARY
- NUMERIC
- TIME

prefix 是在语言环境数据中定义的汇编器符号的前缀

name 是语言环境数据的文本名称。

指定数据

要指定语言环境块数据，请按照文档中指定的顺序为该语言环境类型调用宏。例如：

LC_*TYPE*_*function*

其中：

TYPE 是以下项之一：

- CTYPE
- COLLATE
- MONETARY
- NUMERIC
- TIME

function 是一个与语言环境数据相关的特定函数。

指定语言环境数据时，必须为每个相应函数重复调用宏。

指定结尾

要结束语言环境数据块定义，请调用 _end 宏。该宏不使用任何参数。例如：

LC_*TYPE*_end

其中：

TYPE 是以下项之一：

- CTYPE
- COLLATE
- MONETARY
- NUMERIC
- TIME

指定固定语言环境

要编写一个始终返回相同语言环境的固定函数，您可以使用由这些宏定义的 _start 符号名称。第2-45 页的示例 2-2 说明了如何为 CTYPE 语言环境实现此操作。

示例 2-2 固定语言环境

```
    GET rt_locale.s
    AREA my_locales, DATA, READONLY
    LC_CTYPE_begin my_ctype_locale, "MyLocale"
    ...                         ; include other LC_CTYPE_xxx macros here
    LC_CTYPE_end
    AREA my_locale_func, CODE, READONLY
_get_lc_ctype FUNCTION
    LDR r0, =my_ctype_locale_start
    BX lr
    ENDFUNC
```

指定多个语言环境

必须依次声明适于传递给 _findlocale() 函数的连续语言环境块。您必须调用 LC_index_end 宏以结束语言环境块序列。示例 2-3 说明了如何为 CTYPE 语言环境实现此操作。

示例 2-3 多个语言环境

```
    GET rt_locale.s
    AREA my_locales, DATA, READONLY
my_ctype_locales
    LC_CTYPE_begin my_first_ctype_locale, "MyLocale1"
    ...                         ; include other LC_CTYPE_xxx macros here
    LC_CTYPE_end
    LC_CTYPE_begin my_second_ctype_locale, "MyLocale2"
    ...                         ; include other LC_CTYPE_xxx macros here
    LC_CTYPE_end
    LC_index_end
    AREA my_locale_func, CODE, READONLY
    IMPORT _findlocale
_get_lc_ctype FUNCTION
    LDR r0, =my_ctype_locales
    B _findlocale
    ENDFUNC
```

2.7.4 LC_CTYPE 数据块

用于定义 LC_CTYPE 数据块的宏如下所示：

1. 使用符号名称和语言环境名称调用 LC_CTYPE_begin。有关详细信息，请参阅第2-43 页的*指定开头*。

2. 重复调用 LC_CTYPE_table 以指定 256 个表条目。LC_CTYPE_table 使用单个自变量，并用引号将其引起来。这必须是用逗号分隔的表条目列表。每个表条目描述了 256 个可能的字符之一，它们可以是非法字符 (IL)，也可以是使用任意以下标记的组合集合：

   __S　空白字符
   __P　标点符号字符
   __B　可打印的空白字符
   __L　小写字母
   __U　大写字母
   __N　十进制数字
   __C　控制字符
   __X　十六进制数字字母 A-F 和 a-f
   __A　字母，但不分大小写，如日文片假名。

   注意

   可打印的空白字符被定义为 isprint() 和 isspace() 结果均为真的任何字符。

   不能为与 __N 或 __X 相同的字符指定 __A。

   有关详细信息，请参阅第2-44 页的*指定数据*。

3. 如果需要，可以调用以下一个或两个可选宏：
   - LC_CTYPE_full_wctype. 当此 LC_CTYPE 语言环境处于活动状态时，如果不使用参数调用该宏，则会导致 C99 宽字符 ctype 函数（iswalpha()、iswupper() 等）在整个 Unicode 范围内返回有用的值。如果未指定该宏，宽 ctype 函数则会简单地将前 256 个 wchar_t 值视为与 256 个 char 值相同，而将 wchar_t 范围内的其余值视为包含非法字符。
   - LC_CTYPE_multibyte 将该语言环境定义为多字节字符集。应使用三个参数调用该宏。前两个参数是在多字节字符集和 Unicode 宽字符之间执行转换的函数的名称。最后一个自变量是 C 宏 MB_CUR_MAX 必须为相应字符集指定的值。这两个函数参数具有以下原型：

```
size_t internal_mbrtowc(wchar_t *pwc, char c, mbstate_t *pstate);
size_t internal_wcrtomb(char *s, wchar_t w, mbstate_t *pstate);
```

`internal_mbrtowc()`

将 1 个字节的 c 用作输入，并将更新 pstate 指向的 mbstate_t 的操作作为读取该字节的结果。如果该字节完成多字节字符编码，则会将相应宽字符写入到 pwc 所指向的位置，并返回 1 以指示操作已完成。如果没有完成，则会返回 -2 以指示 mbstate_t 状态更改，并且指示没有输出任何字符。否则，它返回 -1 以指示编码的输入无效。

`internal_wcrtomb()`

将一个宽字符 w 用作输入，并将一些字节写入到 s 所指向的内存中。它返回输出的字节数，或者返回 -1 以指示输入字符在多字节字符集中没有有效的表示形式。

4. 调用 LC_CTYPE_end （不使用参数）以完成语言环境块定义。有关详细信息，请参阅第2-44 页的*指定结尾*。

示例 2-4 显示了 LC_CTYPE 数据块。

示例 2-4 定义 CTYPE 语言环境

```
LC_CTYPE_begin utf8_ctype, "UTF-8"
;
; Single-byte characters in the low half of UTF-8 are exactly
; the same as in the normal "C" locale.
LC_CTYPE_table "__C, __C, __C, __C, __C, __C, __C, __C, __C" ; 0x00-0x08
LC_CTYPE_table "__C+__S, __C+__S, __C+__S, __C+__S, __C+__S"
                                              ; 0x09-0x0D(BS,LF,VT,FF,CR)
LC_CTYPE_table "__C, __C, __C, __C, __C, __C, __C, __C, __C" ; 0x0E-0x16
LC_CTYPE_table "__C, __C, __C, __C, __C, __C, __C, __C, __C" ; 0x17-0x1F
LC_CTYPE_table "__B+__S" ; space
LC_CTYPE_table "__P, __P, __P, __P, __P, __P, __P, __P" ; !"#$%&'(
LC_CTYPE_table "__P, __P, __P, __P, __P, __P, __P" ; )*+,-./
LC_CTYPE_table "__N, __N, __N, __N, __N, __N, __N, __N, __N, __N" ; 0-9
LC_CTYPE_table "__P, __P, __P, __P, __P, __P, __P" ; :;<=>?@
LC_CTYPE_table "__U+__X, __U+__X, __U+__X, __U+__X, __U+__X, __U+__X" ; A-F
LC_CTYPE_table "__U, __U, __U, __U, __U, __U, __U, __U, __U, __U" ; G-P
LC_CTYPE_table "__U, __U, __U, __U, __U, __U, __U, __U, __U, __U" ; Q-Z
LC_CTYPE_table "__P, __P, __P, __P, __P, __P" ; [\]^_`
LC_CTYPE_table "__L+__X, __L+__X, __L+__X, __L+__X, __L+__X, __L+__X" ; a-f
LC_CTYPE_table "__L, __L, __L, __L, __L, __L, __L, __L, __L, __L" ; g-p
LC_CTYPE_table "__L, __L, __L, __L, __L, __L, __L, __L, __L, __L" ; q-z
LC_CTYPE_table "__P, __P, __P, __P" ; {|}~
LC_CTYPE_table "__C" ; 0x7F
```

```
;
; Nothing in the top half of UTF-8 is valid on its own as a
; single-byte character, so they are all illegal characters (IL).
LC_CTYPE_table "IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL"
LC_CTYPE_table "IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL"
LC_CTYPE_table "IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL"
LC_CTYPE_table "IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL"
LC_CTYPE_table "IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL"
LC_CTYPE_table "IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL"
LC_CTYPE_table "IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL"
LC_CTYPE_table "IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL,IL"
;
; The UTF-8 ctype locale wants the full version of wctype.
LC_CTYPE_full_wctype
;
; UTF-8 is a multi-byte locale, so we must specify some
; conversion functions. MB_CUR_MAX is 6 for UTF-8 (the lead
; bytes 0xFC and 0xFD are each followed by five continuation
; bytes).
;
; The implementations of the conversion functions are not
; provided in this example.
;
IMPORT  utf8_mbrtowc
IMPORT  utf8_wcrtomb
LC_CTYPE_multibyte utf8_mbrtowc, utf8_wcrtomb, 6
LC_CTYPE_end
```

2.7.5 LC_COLLATE 数据块

用于定义 LC_COLLATE 数据块的宏如下所示：

1. 使用符号名称和语言环境名称调用 LC_COLLATE_begin。有关详细信息，请参阅第2-43 页的*指定开头*。

2. 调用以下备选宏之一：

   - 重复调用 LC_COLLATE_table 以指定 256 个表条目。LC_COLLATE_table 使用单个自变量，并用引号将其引起来。这必须是用逗号分隔的表条目列表。每个表条目描述了 256 个可能的字符之一，它们可以是指示其排序顺序位置的数字。例如，如果要将字符 A 排在 B 前面，则表中的条目 65 （与 A 相对应）必须小于条目 66 （与 B 相对应）。

   - 不使用参数调用 LC_COLLATE_no_table。这表示归类顺序与字符串比较顺序相同。因此， strcoll() 和 strcmp() 是相同的。

   有关详细信息，请参阅第2-44 页的*指定数据*。

3. 调用 LC_COLLATE_end （不使用参数）以完成语言环境块定义。有关详细信息，请参阅第2-44 页的*指定结尾*。

示例 2-5 显示了 LC_COLLATE 数据块。

示例 2-5 定义 COLLATE 语言环境

```
LC_COLLATE_begin iso88591_collate, "ISO8859-1"
LC_COLLATE_table "0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07"
LC_COLLATE_table "0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f"
LC_COLLATE_table "0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17"
LC_COLLATE_table "0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f"
LC_COLLATE_table "0x20, 0x21, 0x22, 0x23, 0x24, 0x25, 0x26, 0x27"
LC_COLLATE_table "0x28, 0x29, 0x2a, 0x2b, 0x2c, 0x2d, 0x2e, 0x2f"
LC_COLLATE_table "0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37"
LC_COLLATE_table "0x38, 0x39, 0x3a, 0x3b, 0x3c, 0x3d, 0x3e, 0x3f"
LC_COLLATE_table "0x40, 0x41, 0x49, 0x4a, 0x4c, 0x4d, 0x52, 0x53"
LC_COLLATE_table "0x54, 0x55, 0x5a, 0x5b, 0x5c, 0x5d, 0x5e, 0x60"
LC_COLLATE_table "0x67, 0x68, 0x69, 0x6a, 0x6b, 0x6c, 0x71, 0x72"
LC_COLLATE_table "0x73, 0x74, 0x76, 0x79, 0x7a, 0x7b, 0x7c, 0x7d"
LC_COLLATE_table "0x7e, 0x7f, 0x87, 0x88, 0x8a, 0x8b, 0x90, 0x91"
LC_COLLATE_table "0x92, 0x93, 0x98, 0x99, 0x9a, 0x9b, 0x9c, 0x9e"
LC_COLLATE_table "0xa5, 0xa6, 0xa7, 0xa8, 0xaa, 0xab, 0xb0, 0xb1"
LC_COLLATE_table "0xb2, 0xb3, 0xb6, 0xb9, 0xba, 0xbb, 0xbc, 0xbd"
LC_COLLATE_table "0xbe, 0xbf, 0xc0, 0xc1, 0xc2, 0xc3, 0xc4, 0xc5"
LC_COLLATE_table "0xc6, 0xc7, 0xc8, 0xc9, 0xca, 0xcb, 0xcc, 0xcd"
LC_COLLATE_table "0xce, 0xcf, 0xd0, 0xd1, 0xd2, 0xd3, 0xd4, 0xd5"
LC_COLLATE_table "0xd6, 0xd7, 0xd8, 0xd9, 0xda, 0xdb, 0xdc, 0xdd"
LC_COLLATE_table "0xde, 0xdf, 0xe0, 0xe1, 0xe2, 0xe3, 0xe4, 0xe5"
LC_COLLATE_table "0xe6, 0xe7, 0xe8, 0xe9, 0xea, 0xeb, 0xec, 0xed"
LC_COLLATE_table "0xee, 0xef, 0xf0, 0xf1, 0xf2, 0xf3, 0xf4, 0xf5"
LC_COLLATE_table "0xf6, 0xf7, 0xf8, 0xf9, 0xfa, 0xfb, 0xfc, 0xfd"
LC_COLLATE_table "0x42, 0x43, 0x44, 0x45, 0x46, 0x47, 0x48, 0x4b"
LC_COLLATE_table "0x4e, 0x4f, 0x50, 0x51, 0x56, 0x57, 0x58, 0x59"
LC_COLLATE_table "0x77, 0x5f, 0x61, 0x62, 0x63, 0x64, 0x65, 0xfe"
LC_COLLATE_table "0x66, 0x6d, 0x6e, 0x6f, 0x70, 0x75, 0x78, 0xa9"
LC_COLLATE_table "0x80, 0x81, 0x82, 0x83, 0x84, 0x85, 0x86, 0x89"
LC_COLLATE_table "0x8c, 0x8d, 0x8e, 0x8f, 0x94, 0x95, 0x96, 0x97"
LC_COLLATE_table "0xb7, 0x9d, 0x9f, 0xa0, 0xa1, 0xa2, 0xa3, 0xff"
LC_COLLATE_table "0xa4, 0xac, 0xad, 0xae, 0xaf, 0xb4, 0xb8, 0xb5"
LC_COLLATE_end
```

2.7.6 LC_MONETARY 数据块

用于定义 LC_MONETARY 数据块的宏如下所示：

1. 使用符号名称和语言环境名称调用 LC_MONETARY_begin。有关详细信息，请参阅第2-43 页的*指定开头*。
2. 按如下方式调用 LC_MONETARY 数据宏：
   a. 使用两个参数调用 LC_MONETARY_fracdigits：lconv 结构中的 frac_digits 和 int_frac_digits。
   b. 使用四个参数调用 LC_MONETARY_positive：p_cs_precedes、p_sep_by_space、p_sign_posn 和 positive_sign。
   c. 使用四个参数调用 LC_MONETARY_negative：n_cs_precedes、n_sep_by_space、n_sign_posn 和 negative_sign。
   d. 使用两个参数调用 LC_MONETARY_currsymbol：currency_symbol 和 int_curr_symbol。
   e. 使用一个参数调用 LC_MONETARY_point：mon_decimal_point。
   f. 使用一个参数调用 LC_MONETARY_thousands：mon_thousands_sep。
   g. 使用一个参数调用 LC_MONETARY_grouping：mon_grouping。

   有关详细信息，请参阅第2-44 页的*指定数据*。
3. 调用 LC_MONETARY_end （不使用参数）以完成语言环境块定义。有关详细信息，请参阅第2-44 页的*指定结尾*。

示例 2-6 显示了 LC_MONETARY 数据块。

示例 2-6 定义 MONETARY 语言环境

```
LC_MONETARY_begin c_monetary, "C"
LC_MONETARY_fracdigits 255, 255
LC_MONETARY_positive 255, 255, 255, ""
LC_MONETARY_negative 255, 255, 255, ""
LC_MONETARY_currsymbol "", ""
LC_MONETARY_point ""
LC_MONETARY_thousands ""
LC_MONETARY_grouping ""
LC_MONETARY_end
```

2.7.7 LC_NUMERIC 数据块

用于定义 LC_NUMERIC 数据块的宏如下所示：

1. 使用符号名称和语言环境名称调用 LC_NUMERIC_begin。有关详细信息，请参阅第2-43 页的*指定开头*。
2. 按如下方式调用 LC_NUMERIC 数据宏：
   a. 使用一个参数调用 LC_NUMERIC_point：lconv 结构中的 decimal_point。
   b. 使用一个参数调用 LC_NUMERIC_thousands：thousands_sep。
   c. 使用一个参数调用 LC_NUMERIC_grouping：grouping。
3. 调用 LC_NUMERIC_end （不使用参数）以完成语言环境块定义。有关详细信息，请参阅第2-44 页的*指定结尾*。

示例 2-7 显示了 LC_NUMERIC 数据块。

示例 2-7 定义 NUMERIC 语言环境

```
LC_NUMERIC_begin c_numeric,  “C”
LC_NUMERIC_point  “.”
LC_NUMERIC_thousands  “ “
LC_NUMERIC_grouping  “ “
LC_NUMERIC_end
```

2.7.8 LC_TIME 数据块

用于定义 LC_TIME 数据块的宏如下所示：

1. 使用符号名称和语言环境名称调用 LC_TIME_begin。有关详细信息，请参阅第2-43 页的*指定开头*。
2. 按如下方式调用 LC_TIME 数据宏：
   a. 调用 7 次 LC_TIME_week_short，为一周中的每一天提供简短名称。星期日为第一天。然后调用 LC_TIME_week_long，并重复此过程以提供长名称。
   b. 调用 12 次 LC_TIME_month_short，为每月中的某些天提供简短名称。然后调用 LC_TIME_month_long，并重复此过程以提供长名称。
   c. 使用两个参数调用 LC_TIME_am_pm，它们分别是表示上午和下午的字符串。

d. 使用三个参数调用 LC_TIME_formats，它们分别是 strftime("%c") 中使用的标准日期/时间格式、标准日期格式 strftime("%x") 和标准时间格式 strftime("%X")。这些字符串必须根据其他更简单的 strftime 基元来定义标准格式。示例 2-8 说明了允许标准日期/时间格式引用其他两种格式。

e. 使用单个字符串调用 LC_TIME_c99format，它是 strftime("%r") 中使用的标准 12 小时时间格式 （在 C99 中定义）。

3. 调用 LC_TIME_end （不使用参数）以完成语言环境块定义。有关详细信息，请参阅第2-44 页的*指定结尾*。

示例 2-8 显示了 LC_TIME 数据块。

示例 2-8 定义 TIME 语言环境

```
LC_TIME_begin c_time,  “C”
LC_TIME_week_short  “Sun”
LC_TIME_week_short  “Mon”
LC_TIME_week_short "Tue"
LC_TIME_week_short "Wed"
LC_TIME_week_short "Thu"
LC_TIME_week_short "Fri"
LC_TIME_week_short "Sat"
LC_TIME_week_long "Sunday"
LC_TIME_week_long "Monday"
LC_TIME_week_long "Tuesday"
LC_TIME_week_long "Wednesday"
LC_TIME_week_long "Thursday"
LC_TIME_week_long "Friday"
LC_TIME_week_long "Saturday"
LC_TIME_month_short "Jan"
LC_TIME_month_short "Feb"
LC_TIME_month_short "Mar"
LC_TIME_month_short "Apr"
LC_TIME_month_short "May"
LC_TIME_month_short "Jun"
LC_TIME_month_short "Jul"
LC_TIME_month_short "Aug"
LC_TIME_month_short "Sep"
LC_TIME_month_short "Oct"
LC_TIME_month_short "Nov"
LC_TIME_month_short "Dec"
LC_TIME_month_long "January"
LC_TIME_month_long "February"
LC_TIME_month_long "March"
LC_TIME_month_long "April"
```

```
LC_TIME_month_long "May"
LC_TIME_month_long "June"
LC_TIME_month_long "July"
LC_TIME_month_long "August"
LC_TIME_month_long "September"
LC_TIME_month_long "October"
LC_TIME_month_long "November"
LC_TIME_month_long "December"
LC_TIME_am_pm "AM", "PM"
LC_TIME_formats "%x %X", "%d %b %Y", "%H:%M:%S"
LC_TIME_c99format "%I:%M:%S %p"
LC_TIME_week_short  “Sat”
LC_TIME_end
```

2.7.9 _get_lconv()

_get_lconv() 使用与数量格式对应的值设置 lconv 结构的组件。

语法

```
void _get_lconv(struct lconv *lc);
```

用法

此 ISO 扩展不使用任何静态数据。如果要构建严格遵循 ISO C 标准的应用程序，请改用 localeconv()。

返回值

使用格式数据填充现有 lconv 结构 *lc*。

2.7.10 localeconv()

localeconv() 依照当前语言环境规则，使用与数量格式对应的值来创建和设置 lconv 结构的组件。

语法

```
struct lconv *localeconv(void);
```

用法

具有 **char** * 类型的结构成员是一些字符串。除 decimal_point 外，其中的任何成员都可以指向空字符串 ""，表示值在当前语言环境中不可用或者长度为 0。

具有 **char** 类型的成员是非负数。任何成员都可以是 CHAR_MAX，表示值在当前语言环境中不可用。

第2-55 页的*lconv 结构*介绍了 lconv 中包含的成员。

返回值

函数返回一个指向填充对象的指针。程序不会修改返回值所指向的结构，但随后对 localeconv() 函数的调用可能会覆盖该结构。另外，调用具有 LC_ALL、LC_MONETARY 或 LC_NUMERIC 类别的 setlocale() 函数时，可能会覆盖该结构的内容。

2.7.11 setlocale()

选择 *category* 和 *locale* 参数指定的相应语言环境。

语法

```
char *setlocale(int category, const char *locale);
```

用法

setlocale() 函数用于更改或查询当前语言环境的部分或全部。*语言环境类别*介绍了每个 category 自变量值产生的影响。*locale* 值 "C" 指定 C 转换的最低环境。*locale* 的空字符串值 "" 指定实现定义的本机环境。在程序启动时，将执行 setlocale(LC_ALL, "C") 的等效函数。

语言环境类别

category 的值是：

LC_COLLATE　影响 strcoll() 的行为。

LC_CTYPE　影响字符处理函数的行为。

LC_MONETARY　影响 localeconv() 返回的货币格式信息。

LC_NUMERIC　影响设置格式的输入/输出函数、字符串转换函数的小数点字符以及 localeconv() 返回的数字格式信息。

LC_TIME　可能会影响 strftime() 的行为。对于当前支持的语言环境，此选项无效。

LC_ALL　影响所有语言环境类别。这是所有语言环境类别的按位搏驍运算。

返回值

如果为 *locale* 分配了一个指向字符串的指针，并且选择的内容有效，则会返回与新语言环境的指定类别相关联的字符串。如果不支持选择的内容，则会返回空指针，并且不会更改语言环境。

如果 *locale* 为空指针，则会导致返回与当前语言环境类别相关联的字符串，并且不会更改语言环境。

如果 *category* 是 LC_ALL，并且最近一次成功的语言环境调用使用的类别不是 LC_ALL，则可能会返回一个复合字符串。在后续调用中与关联类别一起使用时返回的字符串将恢复该程序语言环境部分。程序不会修改返回的字符串，但随后对 setlocale() 的调用可能会覆盖该字符串。

2.7.12 _findlocale()

_findlocale() 搜索语言环境数据库并返回一个指针，该指针指向所请求类别和语言环境的数据块。

语法

```
void const *_findlocale(void const *index, const char *name);
```

返回值

返回一个指向所请求数据块的指针。

2.7.13 lconv 结构

lconv 结构包含数字格式信息。该结构由 _get_lconv() 和 localeconv() 函数填充。

第2-56 页的示例 2-9 显示了 locale.h 中的 lconv 定义。

示例 2-9 lconv 结构

```
struct lconv {
  char *decimal_point;
       /* The decimal point character used to format non monetary quantities */
  char *thousands_sep;
       /* The character used to separate groups of digits to the left of the */
       /* decimal point character in formatted non monetary quantities.      */
  char *grouping;
       /* A string whose elements indicate the size of each group of digits  */
       /* in formatted non monetary quantities. See below for more details.  */
  char *int_curr_symbol;
       /* The international currency symbol applicable to the current locale.*/
       /* The first three characters contain the alphabetic international    */
       /* currency symbol in accordance with those specified in ISO 4217.    */
       /* Codes for the representation of Currency and Funds. The fourth     */
       /* character (immediately preceding the null character) is the        */
       /* character used to separate the international currency symbol from  */
       /* the monetary quantity.                                             */
  char *currency_symbol;
       /* The local currency symbol applicable to the current locale.        */
  char *mon_decimal_point;
       /* The decimal-point used to format monetary quantities.              */
  char *mon_thousands_sep;
       /* The separator for groups of digits to the left of the decimal-point*/
       /* in formatted monetary quantities.                                  */
  char *mon_grouping;
       /* A string whose elements indicate the size of each group of digits  */
       /* in formatted monetary quantities. See below for more details.      */
  char *positive_sign;
       /* The string used to indicate a non negative-valued formatted
*/
       /* monetary quantity.                                                 */
  char *negative_sign;
       /* The string used to indicate a negative-valued formatted monetary   */
       /* quantity.                                                          */
  char int_frac_digits;
       /* The number of fractional digits (those to the right of the         */
       /* decimal-point) to be displayed in an internationally formatted     */
       /* monetary quantities.                                               */
  char frac_digits;
       /* The number of fractional digits (those to the right of the         */
       /* decimal-point) to be displayed in a formatted monetary quantity.   */
  char p_cs_precedes;
       /* Set to 1 or 0 if the currency_symbol respectively precedes or      */
       /* succeeds the value for a non negative formatted monetary quantity. */
  char p_sep_by_space;
       /* Set to 1 or 0 if the currency_symbol respectively is or is not     */
       /* separated by a space from the value for a non negative formatted   */
```

```
      /* monetary quantity.                                              */
  char n_cs_precedes;
      /* Set to 1 or 0 if the currency_symbol respectively precedes or   */
      /* succeeds the value for a negative formatted monetary quantity.  */
  char n_sep_by_space;
      /* Set to 1 or 0 if the currency_symbol respectively is or is not  */
      /* separated by a space from the value for a negative formatted    */
      /* monetary quantity.                                              */
  char p_sign_posn;
      /* Set to a value indicating the position of the positive_sign for a  */
      /* non negative formatted monetary quantity. See below for more
details*/
  char n_sign_posn;
      /* Set to a value indicating the position of the negative_sign for a  */
      /* negative formatted monetary quantity. */
};
```

grouping 和 non_grouping 的元素 （如第2-56 页的示例 2-9 中所示）解释如下：

CHAR_MAX　　不再执行额外的分组。

0　　其余数字重复前面的元素。

其他　　此值是组成当前组的数字个数。将检查下一个元素，以确定当前组左侧的下一组数字的大小。

p_sign_posn 和 n_sign_posn 的值 （如第2-56 页的示例 2-9 中所示）解释如下：

0　　将数量和货币符号用括号括起来。

1　　将符号字符串放在数量和货币符号前面。

2　　将符号字符串放在数量和货币符号后面。

3　　将符号字符串紧靠货币符号前面放置。

4　　将符号字符串紧靠货币符号后面放置。

2.8 调整错误信号、错误处理和程序退出

C 库产生的所有捕获或错误信号都要通过 __raise() 函数实现。可以重新实现此函数或者它使用的低级函数。

小心

IEEE 754 浮点处理标准规定，对异常的缺省响应是继续执行而不进行捕获。可通过调整 fenv.h 中的函数和定义来修改浮点错误处理方式。另请参阅第 4 章 *浮点支持*。

有关与错误相关的函数的详细信息，请参阅 rt_misc.h 包含文件。

表 2-9 中显示了捕获和错误处理函数。有关应用程序初始化和关闭的其他信息，另请参阅第2-31 页的 *调整 C 库以适应新的执行环境*。

表 2-9 捕获和错误处理

函数	说明
_sys_exit()	所有从库中的退出最终都会调用。请参阅第2-59 页的*_sys_exit()*。
errno	是一个用于错误处理的静态变量。请参阅第2-59 页的*errno*。
__rt_errno_addr()	调用以获取 errno 变量的地址。请参阅第2-60 页的*__rt_errno_addr()*。
__raise()	发出信号以指示运行时异常。请参阅第2-60 页的*__raise()*。
__rt_raise()	发出信号以指示运行时异常。请参阅第2-61 页的*__rt_raise()*。
__default_signal_handler()	向用户显示错误指示。请参阅第2-61 页的*__default_signal_handler()*。
_ttywrch()	将字符写入到控制台中。_ttywrch() 的缺省实现采用半主机方式，因此，它使用半主机调用。请参阅第2-62 页的*_ttywrch()*。
__rt_fp_status_addr()	可以调用此函数以获取 fp 状态字的地址。请参阅第2-62 页的*__rt_fp_status_addr()*。

2.8.1 _sys_exit()

库退出函数。所有从库中的退出最终都会调用 _sys_exit()。

语法

```
void _sys_exit(int return_code);
```

用法

此函数不能返回值。可以使用以下任一方法，在更高级别阻止应用程序退出：

- 将 C 库函数 exit() 作为应用程序的一部分实现。如果这样做，则会失去 atexit() 处理和库关闭功能。
- 将 __rt_exit(int n) 函数作为应用程序的一部分实现。如果这样做，则会失去库关闭功能；但在调用 exit() 时或在 main() 返回时，仍会执行 atexit() 处理。

返回值

返回代码仅供参考。实现可能会尝试将其传递到执行环境。

2.8.2 errno

C 库 errno 变量是在库的隐式静态数据区中定义的。该区由 __user_libspace() 指定。它占用建立运行时堆栈的函数所使用的初始堆栈空间的一部分。errno 定义是：

```
(*(volatile int *) __rt_errno_addr())
```

如果要将 errno 放在用户定义的位置，而不是由 __user_libspace() 指定的缺省位置，则可以定义 __rt_errno_addr()。另请参阅第2-6 页的*__user_libspace 静态数据区*。

返回值

缺省实现是 __user_libspace()（返回状态字地址）上的中间代码。C 库标准头文件中给出了适合的缺省定义。

2.8.3 __rt_errno_addr()

当 C 库尝试读取或写入 errno 时，将调用此函数以获取 C 库 errno 变量的地址。此库提供了一个缺省实现。您不太可能需要重新实现此函数。

语法

```
volatile int *__rt_errno_addr(void);
```

2.8.4 __raise()

此函数发出信号以指示运行时异常。

语法

```
int __raise(int signal, int type);
```

其中：

signal 是一个保存信号编号的整数。

type 是一个整数或字符串常数或变量。

用法

此函数调用普通 C 信号机制或缺省信号处理程序。有关详细信息，另请参阅第2-62 页的*_ttywrch()*。

可通过定义以下内容来替换 __raise() 函数：

```
int __raise(int signal, int type);
```

这样，便可绕过 C 信号机制及其数据消耗信号处理程序向量，但会提供与下面基本相同的接口：

```
void __default_signal_handler(int signal, int type);
```

另请参阅第2-11 页的*ARM C 库中的线程安全性*。

返回值

__raise() 返回条件可能有以下三种：

无返回值 处理程序执行较长的跳转或重新启动。

0 信号已被处理。

非零值 调用代码必须将该返回值传递给退出代码。如果 __raise() 返回非零返回代码 *rc*，缺省库实现将调用 _sys_exit(rc)。

2.8.5 __rt_raise()

此函数发出信号以指示运行时异常。

语法

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

其中：

signal 是一个保存信号编号的整数。

type 是一个整数或字符串常数或变量。

用法

可以重新定义此函数以替换库的整个信号处理机制。缺省实现调用 __raise()。有关详细信息，请参阅第2-60 页的*__raise()*。

取决于从 __raise() 中返回的值：

无返回值 处理程序执行较长的跳转或重新启动，并且 __rt_raise() 并未重获控制权。

0 信号已被处理，并且 __rt_raise() 退出。

非零值 如果 __raise() 返回非零返回代码 *rc*，缺省库实现将调用 _sys_exit(rc)。

2.8.6 __default_signal_handler()

此函数处理发出的信号。缺省操作是输出错误消息，然后退出。

语法

```
void __default_signal_handler(int signal, int type);
```

用法

缺省信号处理程序使用 _ttywrch() 输出消息，然后调用 _sys_exit() 退出。可通过定义以下内容替换缺省信号处理程序：

```
void __default_signal_handler(int signal, int type);
```

此接口与 __raise() 相同，但仅在 C 信号处理机制拒绝处理信号时才会调用此函数。

signal.h 中包含所定义的信号的完整列表。有关库所使用的信号，请参阅第2-98 页的表 2-13。

注意

在将来的产品版本中，库所使用的信号可能会发生变化。

2.8.7 _ttywrch()

此函数将一个字符写入到控制台中。控制台可能已被重定向。除非万不得已，否则不要将此函数用作错误处理例程。

语法

```
void __ttywrch(int ch);
```

用法

此函数的缺省执实现采用半主机方式。

即使没有其他输入/输出，也可以重新定义此函数或 __raise()。例如，它可能会将错误消息写入保存在非易失存储器内的日志中。

2.8.8 __rt_fp_status_addr()

此函数返回浮点状态字的地址。

语法

```
unsigned *__rt_fp_status_addr(void);
```

用法

如果未定义 __rt_fp_status_addr()，则会使用 C 库中的缺省实现。值是在 __rt_lib_init() 调用 _fp_init() 时初始化的。fenv.h 中列出了状态字常数。缺省 fp 状态是 0。

另请参阅第2-11 页的*ARM C 库中的线程安全性*。

2.9 调整存储管理

本节介绍了在调整内存管理时可定义的 rt_heap.h 函数。它还介绍了两个可从堆实现中调用的辅助函数。

有关与内存相关的函数的详细信息，请参阅 rt_heap.h 和 rt_memory.s 包含文件。

2.9.1 避免使用 ARM 提供的堆和使用堆的函数

如果开发的嵌入式系统只有有限的 RAM 或有其自己的堆管理功能 （如操作系统），则可能需要使用不对堆区进行定义的系统。要避免使用堆，您可以执行以下任一操作：

- 在您自己的应用程序中重新实现这些函数
- 按某种方式编写应用程序，以使其不调用任何使用堆的函数。

可以在代码中引用 __use_no_heap 或 __use_no_heap_region 符号，以确保不从 ARM 库中链接使用堆的函数。只需在应用程序中导入一次这些符号即可，例如，使用下列任一方法导入：

- 汇编语言中的 IMPORT __use_no_heap
- C 中的 #pragma import(__use_no_heap)。

如果包含使用堆的函数，并且还引用了 __use_no_heap 或 __use_no_heap_region，链接器将报告错误。例如，以下示例代码导致显示链接器错误：

```
#include <stdio.h>
#include <stdlib.h>
#pragma import(__use_no_heap)

void main()
{
    char *p = malloc(256);
    ...
}
```

```
Error: L6915E: Library reports error: __use_no_heap was requested, but malloc
was referenced
```

要找出使用堆的对象，请使用 --verbose --list=out.txt 进行链接，搜索输出以查找相关符号 （此处为 malloc），然后找出引用该符号的对象。

__use_no_heap 可防止使用 malloc()、 realloc()、 free() 以及使用这些函数的任何函数。例如， calloc() 和其他 stdio 函数。

__use_no_heap_region 与 __use_no_heap 具有相同的特性，但除此之外，还可防止使用堆内存区的其他情况。例如，如果将 main() 声明为使用参数的函数，则将使用堆区收集 argc 和 argv。

2.9.2 malloc 支持

malloc()、realloc()、calloc() 和 free() 均建立在堆抽象数据类型的基础上。可以在 Heap1 和 Heap2 之间进行选择，它们均提供了堆实现。

malloc()、realloc() 和 calloc() 的缺省实现维护一个 8 字节对齐的堆。

Heap1：标准堆实现

Heap1 （缺省实现）实现最小且最简单的堆管理器。堆是作为单链接可用块列表 （按地址升序排列）进行管理的。分配策略是地址优先匹配。

此实现具有较低的开销，但 malloc() 或 free() 的开销随可用块数量的增加呈线性增长。可分配的最小块是 4 个字节，并且产生 4 个字节的附加开销。如果希望得到 100 个以上的未分配块，则建议使用 Heap2。

Heap2：可选堆实现

Heap2 提供了较为紧凑的实现方式，malloc() 或 free() 的开销随可用块数量的增加呈对数增长。分配策略是地址优先匹配。可分配的最小块是 12 个字节，并产生 4 个字节的附加开销。

如果在存在数百个可用块的情况下需要大致恒定的性能 （不会随时间的推移而发生变化），建议使用 Heap2。要选择可选的标准实现，请使用以下任一方法：

- 汇编语言中的 IMPORT __use_realtime_heap
- C 中的 #pragma import(__use_realtime_heap)。

使用 Heap2

Heap2 实时堆实现必须知道堆占用的最大地址空间。地址范围越小，算法越有效。

缺省情况下，将堆的范围设置为 16 MB，从堆的起点 （定义为 __rt_initial_stackheap() 或 __rt_heap_extend() 为堆管理器分配的第一个内存块的起点）开始。

堆范围是由以下代码设置的：

```
struct __heap_extent {
  unsigned base, range;
};

__value_in_regs struct __heap_extent __user_heap_extent(
             unsigned defaultbase, unsigned defaultsize);
```

__user_heap_extent() 的函数原型位于 rt_misc.h 中。

Heap1 算法不需要在堆范围内设置边界，因此，它根本不会调用此函数。

如果以下条件成立，则必须重新定义 __user_heap_extent()：

- 需要的堆占用的地址空间超过 16 MB
- 内存模型可提供一个内存块，其地址低于提供的第一个内存块的地址。

如果事先知道堆的地址空间范围很小，则不需要重新定义 __user_heap_extent()，但这样做会加快堆算法的速度。

如果未定义此例程，则输入参数是所使用的缺省值。例如，可以将缺省基址值保持不变，而只调整大小。

注意

返回的大小字段必须是 2 的幂。库不检查这一情况；如果不满足此要求，则会以意外方式失败。如果返回的大小为 0，则将堆范围设置为 4 GB。

从裸机 C 中使用堆实现

要在未定义 main() 函数并且未初始化 C 库的应用程序中使用堆实现，请执行以下操作：

1. 调用 _init_alloc(*base*, *top*) 以定义要作为堆管理的内存的底部和顶部。
2. 定义 unsigned __rt_heap_extend(**unsigned** size, **void** **block) 函数以处理用于在堆满时扩展堆的调用。

alloca()

alloca() 的行为与 malloc() 相同，只不过 alloca() 具有自动垃圾回收功能（请参阅第2-106 页的*alloca()*）。

2.10 调整运行时内存模型

本节介绍以下内容：

- C 库对可写内存的管理，如静态数据、堆和堆栈
- 可重新定义以更改可写内存管理方式的函数。

有关详细信息，请参阅 《开发指南》中*第3-13 页的放置堆栈和堆*。

2.10.1 内存模型

您可以选择以下任一内存模型：

单内存区

堆栈从内存区顶部向下增长。堆从内存区底部向上增长。这是缺省设置。由堆管理的内存从来不会缩减。不能将通过调用 `free()` 释放的堆内存再次用于其他用途。

双内存区

一个内存区用于堆栈，另一个内存区用于堆。堆区大小可以是零。堆栈区可以位于分配的内存中，也可以从执行环境中继承。

要使用双区模型而不是缺省的单区模型，请使用以下任一方法：

- 汇编语言中的 `IMPORT __use_two_region_memory`
- C 中的 `#pragma import(__use_two_region_memory)`。

注意

如果使用双区内存模型，并且未提供任何堆内存，则无法调用 `malloc()`、使用 `stdio` 或获取 `main()` 的命令行参数。

如果将堆区大小设置为 0，并且将 `__user_heap_extend()` 定义为可扩展堆的函数，则会在需要时创建堆。

有关如何在使用堆或堆区时发出警告消息的详细信息，请参阅第2-64 页的*调整存储管理*中的 `__use_no_heap` 说明。

2.10.2 控制运行时内存模型

要修改堆和堆栈管理器的行为，可以使用下列任意方法：

- 使用分散加载描述文件
- 重新定义 `__user_setup_stackheap()` 和 `__user_heap_extend()`

- 定义用于指定初始堆栈指针以及堆的开头和末尾的符号。

有关分散加载描述文件的信息，请参阅 《链接器用户指南》中的第 5 章 *使用分散加载描述文件*。

__user_setup_stackheap() 设置并返回初始堆栈和堆的位置。__user_heap_extend() 返回附加内存块以供堆使用。在下列情况下，只需重新定义这些函数即可：

- 未使用分散加载描述文件
- 不需要这些函数的缺省实现所使用的值
- 未定义用于指定初始堆栈指针以及堆的开头和末尾的符号 __initial_sp、__heap_base 和 __heap_limit。

注意

- 您可能会看到 __user_initial_stackheap() 而不是 __user_setup_stackheap()。如果您维护的是旧式源代码，则最可能会发生这种情况，因为 __user_initial_stackheap() 是早期版本的实现。
- 建议您将源代码中出现的所有 __user_initial_stackheap() 都替换为 __user_setup_stackheap()。

__user_libspace 为库提供了隐藏静态数据。在库初始化过程中，还会将静态数据区用作堆栈。此函数通常不需要重新实现。请参阅第2-40 页的*调整静态数据访问*。

有关重新定义 __user_setup_stackheap() 的详细信息，请参阅第2-69 页的 *__user_setup_stackheap()*。

有关重新定义 __user_heap_extend() 的详细信息，请参阅第2-73 页的 *__user_heap_extend()*。

有关定义符号 __initial_sp、__heap_base 和 __heap_limit 的信息，请参阅第3-6 页的*创建堆栈*和第3-7 页的*创建堆*。standardlib 和 microlib 均支持这种控制运行时内存模型的方法。

2.10.3 编写您自己的内存模型

如果提供的内存模型不能满足需要，您可以编写自己的内存模型。内存模型必须定义第2-69 页的表 2-10 中所描述的函数。所有函数都是 ARM 状态函数。如果需要，库可负责管理从 Thumb 状态的转变。位于 ...\include 目录中的 rt_memory.s 提供了一个不完整的模型原型实现。

可以从此原型入手来创建您自己的实现。

表 2-10 内存模型函数

函数	说明
__rt_stackheap_init()	创建并初始化堆栈指针。返回一个内存区以用作初始堆。一个汇编器级指令。
__rt_heap_extend()	返回新内存块以添加到堆中。请参阅第2-74 页的 *__rt_heap_extend()*。

2.10.4 __user_setup_stackheap()

__user_setup_stackheap() 设置并返回初始堆栈和堆的位置。

与 __user_initial_stackheap() 不同， __user_setup_stackheap() 用于如下系统：应用程序启动时使用已是正确的 sp (r13) 值，例如， Cortex-M3。要使用 sp，请实现 __user_setup_stackheap() 以设置 r0、 r2 和 r3 （对于双区模型）并返回。对于单区模型，仅设置 r0。

调用 __user_setup_stackheap() 时， sp 的值与在其应用程序入口处的值相同。如果该值无效， __user_setup_stackheap() 在使用任何堆栈之前必须更改 sp 的值。

使用 __user_setup_stackheap() 而不是 __user_initial_stackheap() 可减小代码大小，这是因为不需要设置临时堆栈。

__user_setup_stackheap() 返回：

- r0 中的堆基址
- sp 中的堆栈基址
- r2 中的堆限制
- r3 中的堆栈限制。

注意

必须在汇编器中重新实现 __user_setup_stackheap()。

2.10.5 __user_initial_stackheap()

如果您使用的是旧式源代码，则可能会看到 __user_initial_stackheap()。这是旧函数，支持它只是为了与旧式源代码向后兼容。其当前等效项是 __user_setup_stackheap()。

从 RVCT v2.x 及更低版本移植到 RVCT v4.0

在 RVCT 2.x 及更低版本中，__user_initial_stackheap() 的缺省实现使用符号 Image$$ZI$$Limit 的值。如果链接器使用分散加载描述文件（使用 --scatter 命令行选项指定），则不会定义该符号；因此，如果使用的是分散加载描述文件，则必须重新实现 __user_initial_stackheap()，否则，链接步骤将会失败。

另外，您也可以使用 __user_setup_stackheap()（而不是 __user_initial_stackheap()）来升级源代码。

从 RVCT v3 移植到 RVCT v4.0

在 RVCT v3.x 和更高版本中，库包含更多的 __user_initial_stackheap() 实现，并且可通过分散加载描述文件中提供的信息自动选择正确的实现。这意味着，如果使用的是分散加载文件，则不需要重新实现该函数。有关详细信息，请参阅第2-71 页的*使用分散加载描述文件*。

语法

```
extern __value_in_regs struct __user_initial_stackheap
__user_initial_stackheap(unsigned R0, unsigned SP, unsigned R2, unsigned SL);
```

用法

__user_initial_stackheap() 返回：

- r0 中的堆基址
- r1 中的堆栈基址，即堆栈区中的最高地址
- r2 中的堆限制
- r3 中的堆栈限制，即堆栈区中的最低地址。

如果重新实现了此函数，则必须满足下列条件：

- 最多使用堆栈中的 88 个字节
- 不能破坏 r12 (ip) 以外的寄存器
- 保持堆的 8 字节对齐方式。

对于缺省单区模型，将忽略 r2 和 r3 中的值，并且 r0 和 r1 之间的所有内存始终可供堆使用。对于双区模型，堆限制是由 r2 设置的，堆栈限制是由 r3 设置的。

调用 __main() 时的 sp (r13) 值将作为 r1 中的自变量进行传递。__user_initial_stackheap() 的缺省实现 （使用半主机 SYS_HEAPINFO）是由 sys_stackheap.o 模块中的库提供的。

要创建从执行环境中继承 sp 并且不使用堆的 __user_initial_stackheap() 版本，请将 r0 和 r2 设置为 r3 的值并返回。有关详细信息，请参阅第2-69 页的 *__user_setup_stackheap()*。

rt_misc.h 中的 __initial_stackheap 的定义是：

```
struct __initial_stackheap{
    unsigned heap_base, stack_base, heap_limit, stack_limit;
};
```

注意

stack_base 的值比堆栈使用的最高地址大 0x1，这是因为使用的是满降序堆栈。

有关该函数的重新实现示例，请参阅示例目录。

返回值

r0 到 r3 中返回的值取决于是使用单区内存模型，还是使用双区内存模型：

单区 (r0,r1) 是单个堆栈和堆区。r1 大于 r0，并忽略 r2 和 r3。

双区 (r0, r2) 是初始堆，(r3, r1) 是初始堆栈。r2 大于或等于 r0。r3 小于 r1。

使用分散加载描述文件

__user_initial_stackheap() 的缺省实现使用 Image$$ZI$$Limit 符号的值。如果链接器使用分散加载描述文件，则不会定义此符号。不过，C 库提供了替代实现，可通过分散加载描述文件中的信息来使用这些实现。

自动选择单区模型

在分散加载描述文件中定义一个特殊执行区，即 ARM_LIB_STACKHEAP。该区具有 EMPTY 属性。

这导致库选择一个 __user_initial_stackheap() 实现，它将该区用作复合堆/堆栈区。它使用 Image$$ARM_LIB_STACKHEAP$$Base 和 Image$$ARM_LIB_STACKHEAP$$ZI$$Limit 符号的值。

自动选择双区模型

在分散加载描述文件中定义两个特殊执行区：ARM_LIB_HEAP 和 ARM_LIB_STACK。两个区均具有 EMPTY 属性。

这导致库选择一个使用以下符号值的 __user_initial_stackheap() 实现：
Image$$ARM_LIB_HEAP$$Base、 Image$$ARM_LIB_HEAP$$ZI$$Limit、
Image$$ARM_LIB_STACK$$Base 和 Image$$ARM_LIB_STACK$$ZI$$Limit。

示例 2-10 显示了一个用于定义 ARM_LIB_HEAP 和 ARM_LIB_STACK 的分散加载描述文件示例。（它是在主示例目录 *install_directory*\RVDS\Examples 中作为 Cortex-M3.scat 提供的。）

示例 2-10 ARM_LIB_HEAP 和 ARM_LIB_STACK 分散加载描述

```
FLASH_LOAD 0x0000 0x00200000
{
  ;; Maximum of 256 exceptions (256*4 bytes == 0x400)
  VECTORS 0x0 0x400
  {
    ; Exception table provided by the user in exceptions.c
    exceptions.o (exceptions_area, +FIRST)
  }

  ;; Code is placed immediately (+0) after the previous root region
  ;; (so code region will also be a root region)
  CODE +0
  {
    * (+RO)
  }

  DATA 0x20000000 0x00100000
  {
    * (+RW, +ZI)
  }

  ;; Heap starts at 1MB and grows upwards
  ARM_LIB_HEAP 0x20100000 EMPTY 0x100000-0x8000
  {
  }
  ;; Stack space starts at the end of the 2MB of RAM
  ;; And grows downwards for 32KB
  ARM_LIB_STACK 0x20200000 EMPTY -0x8000
  {
  }
}
```

将从 ARM_LIB_STACKHEAP （对于单区模型）或 ARM_LIB_STACK （对于双区模型）中对 sp 进行相应的初始化。

错误消息

如果使用分散文件，而没有指定任何特殊区名称，并且没有重新实现 `__user_initial_stackheap()`，库将生成一条错误消息。

2.10.6 __user_heap_extend()

可以定义此函数以返回供堆使用的附加内存块 （与初始块是分开的）。如果已定义，此函数必须返回 8 字节对齐堆扩展块的大小和基址。

语法

```
extern unsigned __user_heap_extend(int 0, void **base, unsigned
requested_size);
```

用法

此函数没有缺省实现。如果定义此函数，它必须具有以下特征：

- 返回的大小必须是以下任一值：
  - 至少是所请求大小的 8 字节倍数
  - 0，表示无法满足请求。
- 大小按字节计算。
- 此函数仅受 AAPCS 约束的限制。
- 在入口处，第一个自变量始终为 0，可以将其忽略。基址是在保存此自变量的寄存器中返回的。
- 返回的基址必须在 8 字节的边界上对齐。

返回值

此函数将指向至少为请求大小的块的指针放在 *base 中，并返回该块的大小。如果无法返回此类块，则返回 0，在这种情况下，永远不会使用存储在 *base 中的值。

2.10.7 __user_heap_extent()

如果已定义，此函数将返回堆的基址和最大范围。

语法

```
extern __value_in_regs struct __heap_extent __user_heap_extent(unsigned
ignore1, unsigned ignore2);
```

另请参阅第2-65 页的*malloc 支持*。

用法

此函数没有缺省实现。函数不使用 *ignore1* 和 *ignore2* 参数的值。

2.10.8 __rt_stackheap_init()

此函数设置堆栈指针，并返回一个内存区以用作初始堆。从库初始化代码中调用此函数。

从该函数返回时，SP 必须指向堆栈区域的顶部，r0 必须指向堆区域的底部，而 r1 必须指向堆区域的限制。

根据需要，将从 __user_perproc_libspace 区为用户定义的内存模型 （即 __rt_stackheap_init() 和 __rt_heap_extend()）分配 16 字节的存储。该模型通过调用 __rt_stackheap_storage() 以返回一个指向其 16 字节区的指针，来访问此存储。

2.10.9 __rt_heap_extend()

如果可能，此函数将返回一个要添加到堆中的新 8 字节对齐内存块。如果重新实现 __rt_stackheap_init()，则必须重新实现此函数。rt_memory.s 中提供了一个不完整的原型实现。

语法

```
extern unsigned __rt_heap_extend(unsigned size, void **block);
```

用法

调用约定是普通 AAPCS。在入口处，r0 是要添加的块的最小大小，r1 用于保存指向基址存储位置的指针。

缺省实现具有以下特征：

- 返回的大小是以下任一值：
  — 至少是所请求大小的 8 字节倍数

— 0，表示无法满足请求。

- 返回的基址在 8 字节的边界上对齐。
- 大小按字节计算。
- 此函数仅受 AAPCS 约束的限制。

返回值

如果有足够的可用堆内存，缺省实现将扩展堆。如果无法扩展，它将调用 `__user_heap_extend()`（如果已实现），请参阅第2-73 页的*__user_heap_extend()*。在出口处，`r0` 为获取的块的大小；如果无法获取任何块，则为 0。`r1` 在入口处指向的内存位置包含块的基址。

2.11 调整输入/输出函数

高级输入/输出函数 （如 fscanf() 和 fprintf()）以及 C++ 对象 std::cout 与目标无关。不过，这些高级函数通过调用与目标相关的低级函数来执行输入/输出。要重定向输入/输出的目标，您可以避免使用这些高级函数，或者重新定义低级函数。

有关 I/O 函数的详细信息，请参阅 rt_sys.h。

另请参阅第2-5 页的*编写可重入且线程安全的代码*。

2.11.1 对低级函数的依赖性

第2-77 页的表 2-11 显示了高级函数对低级函数的依赖性。如果您定义了自己的低级函数版本，则可以直接使用库中的高级函数版本。fgetc() 使用 __FILE，但 fputc() 使用 __FILE 和 ferror()。

注意

如果使用其任意关联的高级函数，*必须* 提供 __stdin 和 __stdout 的定义。即使其他函数 （如 fgetc() 和 fputc()）的重新实现未引用存储在这些对象中的任何数据，这一点仍会适用。

表关键字：

1. __FILE[1]
2. __stdin[2]
3. __stdout[3]
4. fputc()[4]
5. ferror()[5]
6. fgetc()[6]
7. fgetwc()
8. fputwc()
9. __backspace()[7]

1. 文件结构。
2. __FILE 类型的标准输入对象。
3. __FILE 类型的标准输出对象。
4. 将一个字符输出到文件中。
5. 返回在文件 I/O 期间累积的错误状态。
6. 从文件中获取一个字符。
7. 将文件指针移到上一个字符。请参阅第2-82 页的*重新实现 __backspace()*。

10. __backspacewc()

表 2-11 输入/输出依赖性

高级函数	低级对象									
	1	2	3	4	5	6	7	8	9	10
fgets	x	-	-	-	x	x	-	-	-	-
fprintf	x	-	-	x	x	-	-	-	-	-
fputs	x	-	-	x	-	-	-	-	-	-
fread	x	-	-	-	-	x	-	-	-	-
fscanf	x	-	-	-	-	x	-	-	x	-
fwprintf	x	-	-	-	x	-	-	x	-	-
fwrite	x	-	-	x	-	-	-	-	-	-
fwscanf	x	-	-	-	-	-	x	-	-	x
getchar	x	x	-	-	-	x	-	-	-	-
gets	x	x	-	-	x	x	-	-	-	-
getwchar	x	x	-	-	-	-	x	-	-	-
perror	x	-	x	x	-	-	-	-	-	-
printf	x	-	x	x	x	-	-	-	-	-
putchar	x	-	x	x	-	-	-	-	-	-
puts	x	-	x	x	-	-	-	-	-	-
putwchar	x	-	x	-	-	-	-	x	-	-
scanf	x	x	-	-	-	x	-	-	x	-
vfprintf	x	-	-	x	x	-	-	-	-	-
vfscanf	x	-	-	-	-	x	-	-	x	-
vfwprintf	x	-	-	-	x	-	-	x	-	-
vfwscanf	x	-	-	-	-	-	x	-	-	x
vprintf	x	-	x	x	x	-	-	-	-	-
vscanf	x	x	-	-	-	x	-	-	x	-

表 2-11 输入/输出依赖性 （续）

高级函数	低级对象									
vwprintf	x	-	x	-	x	-	-	x	-	-
vwscanf	x	x	-	-	-	-	x	-	-	x
wprintf	x	-	x	-	x	-	-	x	-	-
wscanf	x	x	-	-	-	-	x	-	-	x

有关低级函数的语法，请参阅 ISO C 参考。

注意

如果选择重新实现 fgetc()、fputc() 和 __backspace()，请注意 fopen() 和相关函数使用的是 ARM 的 __FILE 结构布局。如果您定义了自己的 __FILE 版本，可能还需要重新实现 fopen() 和相关函数。

printf 系列

printf 系列包含 _printf()、printf()、_fprintf()、fprintf()、vprintf() 和 vfprintf()。所有这些函数不透明地使用 __FILE，并且仅依赖于 fputc() 和 ferror() 函数。除了不能设置浮点值的格式外，printf() 和 fprintf() 函数与 printf() 和 fprintf() 是相同的。

_printf(...) 格式的标准输出函数等效于：

```
fprintf(& __stdout, ...)
```

其中，__stdout 具有 __FILE 类型。

scanf 系列

scanf() 系列包含 scanf() 和 fscanf()。这些函数仅依赖于 fgetc()、__FILE 和 __backspace() 函数。请参阅第2-82 页的*重新实现 __backspace()*。

scanf(...) 格式的标准输入函数等效于：

```
fscanf(& __stdin, ...)
```

其中，__stdin 具有 __FILE 类型。

fwrite()、 fputs() 和 puts()

如果您定义了自己的 __FILE 版本、自己的 fputc() 和 ferror() 函数以及 __stdout 对象，则可以直接使用库中的所有 printf() 系列、 fwrite()、 fputs()、 puts() 以及 C++ 对象 std::cout，而无需进行更改。示例 2-11 和第2-80 页的示例 2-12 说明了如何实现此操作。如果需要实际进行文件处理，应考虑修改系统例程。

您不需要重新实现这些示例中所显示的每个函数。只需重新实现有此需要的函数即可。

示例 2-11 重定向 printf() 的目标

```
#include <stdio.h>

struct __FILE
{
  int handle;

  /* Whatever you require here. If the only file you are using is */
  /* standard output using printf() for debugging, no file handling */
  /* is required. */
};

/* FILE is typedef' d in stdio.h. */

FILE __stdout;

int fputc(int ch, FILE *f)
{
  /* Your implementation of fputc(). */
  return ch;
}
int ferror(FILE *f)
{
  /* Your implementation of ferror(). */
  return 0;
}
void test(void)
{
  printf( "Hello world\n" );
}
```

请注意 fputc() 的端标记。fputc() 使用了 int 参数，但只包含一个字符。该字符是位于整数变量的最高字节还是最低字节将取决于端标记。下面的代码示例避免了端标记问题：

```
extern void sendchar(char *ch);

int fputc(int ch, FILE *f)
{
  /* example: write a character to an LCD */
  char tempch = ch; // temp char avoids endianness issue
  sendchar(&tempch);
  return ch;
}
```

另请参阅主示例目录的 ...\emb_sw_dev\source\retarget.c 中的实现。

示例 2-12 重定向 cout 的目标

```
File 1: Reimplement any functions that require reimplementation.

#include <stdio.h>

namespace std {

  struct __FILE
  {
    int handle;

    /* Whatever you require here. If the only file you are using is */
    /* standard output using printf() for debugging, no file handling */
    /* is required. */
  };

  FILE __stdout;
  FILE __stdin;
  FILE __stderr;

  int fgetc(FILE *f)
  {
    /* Your implementation of fgetc(). */
    return 0;
  };
  int fputc(int c, FILE *stream)
  {
    /* Your implementation of fputc(). */
  }
  int ferror(FILE *stream)
  {
```

```
    /* Your implementation of ferror(). */
  }
  long int ftell(FILE *stream)
  {
    /* Your implementation of ftell(). */
  }
  int fclose(FILE *f)
  {
    /* Your implementation of fclose(). */
    return 0;
  }
  int fseek(FILE *f, long nPos, int nMode)
  {
    /* Your implementation of fseek(). */
    return 0;
  }
  int fflush(FILE *f)
  {
    /* Your implementation of fflush(). */
    return 0;
  }
}
```

File 2: Print "Hello world" using your reimplemented functions.

```
#include <stdio.h>
#include <iostream>
using namespace std;

int main()
{
  cout <<  "Hello world\n" ;
  return 0;
}
```

缺省情况下，fread() 和 fwrite() 调用快速块输入/输出函数，这些函数是 ARM 流实现的一部分。如果您定义自己的 __FILE 结构而不是使用 ARM 流实现，fread() 和 fwrite() 将调用 fgetc()，而不是调用块输入/输出函数。

fread()、 fgets() 和 gets()

fread()、fgets() 和 gets() 函数是作为 fgetc() 和 ferror() 中的循环实现的。每个函数都不透明地使用 FILE 自变量。

如果您提供自己的 __FILE、__stdin （用于 gets()）、fgetc() 和 ferror() 实现，则可以直接使用库中的这些函数以及 C++ 对象 std::cin。

重新实现 __backspace()

__backspace() 函数由 scanf 函数系列使用。切勿直接调用此函数，如果在 fgetc() 级别重定向 stdio 布局的目标，则可以重新实现该函数。

语义为：

```
int __backspace(FILE *stream);
```

必须先从流中读出字符，然后再调用 __backspace(stream)。例如，不能在写入、搜索后调用它，或者在打开文件后立即调用它。它向流中返回从流中读取的最后一个字符，以便下一读取操作能够再次从流中读取同一字符。这意味着，scanf 从流中读取一个字符，但它不是所需的字符 （即，它终止 scanf 操作），下一个从流中读取数据的函数将正确读取该字符。

__backspace 与 ungetc() 是分开的。这可确保在 scanf 函数系列结束后推回单个字符。

__backspace() 返回的值是 0 （成功）或 EOF （失败）。它仅在使用不当时才返回 EOF，例如，没有从流中读取任何字符。在正确使用时， __backspace() 一定会始终返回 0，因为 scanf 函数系列并不检查错误的返回结果。

__backspace() 和 ungetc() 之间的交互是：

- 如果对流应用 __backspace()，然后使用 ungetc() 将一个字符返回到相同的流中，则随后的 fgetc() 调用必须先返回由 ungetc() 返回的字符，然后返回由 __backspace() 返回的字符。
- 如果使用 ungetc() 将一个字符返回到流中，随后使用 fgetc() 读取该字符，然后对此字符应用 backspace，则 fgetc() 读取的下一个字符一定是返回到流中的同一字符。即， __backspace() 操作必须抵消 fgetc() 操作的效果。不过，在调用 __backspace() 之后，不必再调用 ungetc() 即可成功完成操作。
- 以下情况是从不会发生的：使用 ungetc() 将一个字符返回到流中，然后立即将 __backspace() 应用于另一个字符，而中间没有进行任何读取操作。必须先调用 fgetc()，然后才能调用 __backspace()，因此，上述调用顺序是非法的。如果要编写 __backspace() 实现，可以假定下面这种情况永远不会发生：先使用 unget() 将一个字符返回到流中，然后立即使用 __backspace()，而中间没有进行任何读取操作。

2.11.2 与目标相关的输入和输出支持函数

rt_sys.h 定义了 FILEHANDLE 类型。FILEHANDLE 的值由 _sys_open() 返回，并指定主机系统上的一个已打开文件。

与目标相关的输入和输出函数使用半主机。如果重新定义了任何函数，则必须重新定义所有流支持函数。

2.11.3 _sys_open()

此函数打开一个文件。

语法

```
FILEHANDLE _sys_open(const char *name, int openmode)
```

用法

fopen() 和 freopen() 需要使用 _sys_open 函数。如果要使用任何文件输入/输出函数，则又需要使用这两个函数。

openmode 参数是一个位映射，其位通常直接对应于 ISO 模式规范。有关详细信息，请参阅 rt_sys.h。可以创建与目标相关的扩展，但还必须扩展 freopen()。

返回值

如果出现错误，则返回值为 -1。

2.11.4 _sys_close()

此函数关闭以前使用 _sys_open() 打开的文件。

语法

```
int _sys_close(FILEHANDLE fh)
```

用法

如果要使用任何输入/输出函数，则必须定义此函数。

返回值

如果成功，则返回值为 0。非零值表示出现错误。

2.11.5 _sys_read()

此函数将文件内容读取到缓冲区中。

语法

```
int _sys_read(FILEHANDLE fh, unsigned char *buf, unsigned len, int mode)
```

注意

由于历史方面的原因，此处列出了 mode 参数。它不包含任何有用的信息，必须将其忽略。

返回值

返回值是以下内容之一：

- *未* 读取的字符数 （即， *len* - 读取的 *result* 个字符）。
- 错误指示。
- EOF 指示符。EOF 指示涉及在正常结果中设置 0x80000000。与目标无关的代码可处理以下两种情况：

  Early EOF 上次读取文件时返回一些字符以及 EOF 指示符。

  Late EOF 上次读取时仅返回 EOF。

2.11.6 _sys_write()

将缓冲区内容写入到以前使用 _sys_open() 打开的文件中。

语法

```
int _sys_write(FILEHANDLE fh, const unsigned char *buf, unsigned len, int mode)
```

注意

由于历史方面的原因，此处列出了 mode 参数。它不包含任何有用的信息，必须将其忽略。

返回值

返回值是以下内容之一：

- 一个正数，表示*未*写入的字符数（因此，任何非零返回值都表示某种类型的失败操作）
- 一个负数，表示出现错误。

2.11.7 _sys_ensure()

此函数刷新与文件句柄关联的缓冲区。

语法

```
int _sys_ensure(FILEHANDLE fh)
```

用法

_sys_ensure() 调用将刷新与文件句柄 *fh* 相关联的任何缓冲区，并且确保将备份存储介质上的文件保持最新状态。

返回值

如果出现错误，则结果为负数。

2.11.8 _sys_flen()

此函数返回文件的当前长度。

语法

```
long _sys_flen(FILEHANDLE fh)
```

用法

按照 _sys_seek() 的要求将 fseek(..., SEEK_END) 转换为 (..., SEEK_SET) 时，需要使用此函数。

如果将 fseek() 用于不直接支持相对于文件末尾的搜索的基本系统，则必须定义 _sys_flen()。如果基本系统可以相对于文件末尾进行搜索，则可以定义 fseek()，这样就不再需要使用 _sys_flen() 了。

返回值

此函数返回文件 *fh* 的当前长度，或者返回一个负数错误指示符。

2.11.9 _sys_seek()

此函数将文件指针放在与文件开头之间的偏移为 *pos* 的位置。

语法

```
int _sys_seek(FILEHANDLE fh, long pos)
```

用法

此函数将当前读取或写入位置设置为相对于当前文件 *fh* 开头的新位置 *pos*。

返回值

如果没有出现错误，结果为非负数；如果出现错误，结果为负数。

2.11.10 _sys_istty()

此函数确定文件句柄是否指定一个终端。

语法

```
int _sys_istty(FILEHANDLE fh)
```

用法

如果将文件连接到终端设备上，则缺省使用此函数提供未缓冲的行为（未调用 `set(v)buf`）并禁止进行搜索。

返回值

返回值是：

0 没有交互设备。

1 有交互设备。

其他 出现错误。

2.11.11 _sys_tmpnam()

此函数将临时文件的文件编号 *fileno* 转换为唯一的文件名，例如，`tmp0001`。

语法

```
void _sys_tmpnam(char *name, int fileno, unsigned maxlength)
```

用法

如果使用 tmpnam() 或 tmpfile()，则必须定义此函数。

返回值

在 *name* 中返回文件名。

2.11.12 _sys_command_string()

此函数从调用当前应用程序的环境中检索用于调用该应用程序的命令行。

语法

```
char *_sys_command_string(char *cmd, int len)
```

其中：

cmd 是一个指针，它指向可用于存储命令行的缓冲区。不要求将命令行存储在 *cmd* 中。

len 是缓冲区的长度。

用法

库启动代码调用此函数以设置要传递给 main() 的 argv 和 argc。

注意

调用此函数时，不能假定 C 库完全初始化。例如，不能从该函数中调用 malloc()。这是因为 C 库启动序列在完全配置堆之前调用该函数。

返回值

此函数必须返回以下任一内容：

- 如果成功，则返回一个指向命令行的指针。它可以是指向 *cmd* 缓冲区（如果已使用）的指针，也可以是指向命令行的其他存储位置的指针。
- 如果失败，则返回 NULL。

2.11.13 #pragma import(_main_redirection)

在运行时重定向标准输入、输出和错误流时，必须定义此编译指示。有关详细信息，请参阅 《编译器参考指南》中第B-3 页的*环境*。

语法

```
#pragma import(_main_redirection)
```

2.12 调整其他 C 库函数

本节介绍与目标相关的 ISO C 库函数。这些 ISO 标准函数的实现完全取决于目标操作系统。

这些函数的缺省实现是采用半主机方式。即，每个函数都使用半主机。

2.12.1 clock()

这是 time.h 中的标准 C 库时钟函数。

语法

```
clock_t clock(void)
```

用法

如果 clock_t 的单位不是缺省单位百分之一秒，则必须在编译器命令行或您自己的头文件中定义 __CLK_TCK。该定义中的值用于 CLK_TCK 和 CLOCKS_PER_SEC。百分之一秒单位的缺省值是 100。

注意

如果要重新实现 clock()，则还必须重新实现 _clock_init()。

返回值

返回值是无符号整数。

2.12.2 _clock_init()

这是一个用于 clock() 的可选初始化函数。

语法

```
__weak void _clock_init(void)
```

用法

如果 clock() 必须与只读计时器配合使用，则必须提供一个时钟初始化函数。如果已实现，则会从库初始化代码中调用 _clock_init()。

2.12.3 time()

这是 time.h 中的标准 C 库 time() 函数。

语法

```
time_t time(time_t *timer)
```

返回值是当前日历时间的近似值。

返回值

如果日历时间不可用，则返回值 -1。如果 *timer* 不是 NULL 指针，则还会将返回值存储在 *timer* 中。

2.12.4 remove()

这是 stdio.h 中的标准 C 库 remove() 函数。

语法

```
int remove(const char *filename)
```

用法

remove() 导致删除一个文件，其名称为 *filename* 所指向的字符串。随后尝试打开该文件的操作将会失败，除非重新创建了该文件。如果已打开此文件，将由实现定义 remove() 函数的行为。

返回值

如果操作成功，则返回 0；如果失败，则返回非零值。

2.12.5 rename()

这是 stdio.h 中的标准 C 库 rename() 函数。

语法

```
int rename(const char *old, const char *new)
```

用法

rename() 导致将一个文件（其名称为 *old* 所指向的字符串）重命名为 *new* 所指向的字符串。将有效地删除名为 *old* 的文件。在调用 rename() 函数之前，如果名称为 *new* 所指向的字符串的文件已存在，将由实现定义此函数的行为。

返回值

如果操作成功，则返回 0；如果失败，则返回非零值。如果操作返回非零值，并且该文件以前存在，则该文件仍使用其原始名称。

2.12.6 system()

这是 stdlib.h 中的标准 C 库 system() 函数。

语法

```
int system(const char *string)
```

用法

system() 将 *string* 所指向的字符串传递到主机环境中，并由命令处理器以实现定义的方式执行。*string* 可以使用空指针以查明命令处理器是否存在。

返回值

如果自变量是空指针，则仅当命令处理器可用时，此系统函数才会返回非零值。

如果自变量不是空指针，system() 函数将返回由实现定义的值。

2.12.7 getenv()

这是 stdlib.h 中的标准 C 库 getenv() 函数。

语法

```
char *getenv(const char *name)
```

用法

缺省实现返回 NULL，表示没有可用环境信息。您可以重新实现 getenv()。该函数不依赖于其他函数，其他函数也不依赖于该函数。

如果重新定义了该函数，您也可以调用 _getenv_init() 函数。然后， C 库初始化代码在初始化库时 （即进入 main() 之前）调用该函数。

除非您重新定义了 getenv() 中应有的正常行为，否则标准 C 库 getenv() 函数 getenv() 的行为是在主机环境所提供的环境列表中搜索与 *name* 所指向的字符串相匹配的字符串。此环境名称集合和更改环境列表的方法是由实现定义的。

返回值

返回值是一个指针，它指向与匹配的列表成员相关联的字符串。程序不能修改所指向的数组，但后续 getenv() 调用可能会覆盖此数组。

2.12.8 _getenv_init()

这允许 getenv() 的用户版本对其自身进行初始化。

语法

```
void _getenv_init(void)
```

用法

如果定义了此函数， C 库初始化代码在初始化库时将调用此函数，即，在进入 main() 之前。

2.13 选择实时除法

随 ARM 库提供的除法辅助例程可提供很好的整体性能。不过，执行除法操作所需的时间取决于输入值。4 位的商只需要 12 次循环；而 32 位的商则需要 96 次循环。视目标而定，某些应用程序需要在最坏情况下获得更快的循环速度，但会以降低平均性能为代价。为此， ARM 库提供了两个除法例程。

实时例程：

- 执行的循环始终少于 45 次
- 对于较大的商，此例程比标准除法辅助例程快
- 对于典型的商，此例程比标准除法辅助例程慢
- 返回相同结果
- 不需要对周围代码进行任何更改。

可以使用以下任一方法选择实时除法例程，而不是通常更有效的例程：

- 汇编语言中的 `IMPORT __use_realtime_division`
- C 中的 `#pragma import(__use_realtime_division)`。

——注意——

对于 Cortex-M1，库中没有提供实时除法。

2.14 ISO 实现定义

本节介绍了库如何满足 ISO 规范的要求。

2.14.1 ISO C 库实现定义

ISO 规范允许实现者实现某些功能，但要求记录选择的实现。本节介绍了 ARM 库实现。

在通用 ARM C 库中：

- 宏 `NULL` 扩展为整型常数 `0`。
- 如果程序重新定义了保留的外部标识符，当程序与标准库进行链接时，可能会出现错误。如果程序没有与标准库进行链接，则不会诊断出任何错误。
- `__aeabi_assert()` 函数在 `stderr` 中输出有关失败诊断的信息，然后调用 `abort()` 函数：

  `*** assertion failed: expression, file name, line number`

  注意

  `assert` 宏的行为取决于最近出现的 `#include <assert.h>` 的运行条件。有关详细信息，请参阅第2-36 页的*从声明中退出*。

以下函数对 `EOF` (-1) 至 255 （包含这两个值）范围内的字符值进行测试：

- `isalnum()`
- `isalpha()`
- `iscntrl()`
- `islower()`
- `isprint()`
- `isupper()`
- `ispunct()`.

第2-111 页的*库命名约定*中列出了 ISO C 库变体。

数学函数

表 2-12 说明了在提供的参数超出范围时数学函数的响应方式。

表 2-12 数学函数

效恨	条件	返回值	错误编号
acos(x)	abs(x) > 1	QNaN	EDOM
asin(x)	abs(x) > 1	QNaN	EDOM
atan2(x,y)	x =0, y = 0	QNaN	EDOM
atan2(x,y)	x = Inf, y = Inf	QNaN	EDOM
cos(x)	x=Inf	QNaN	EDOM
cosh(x)	溢出	+Inf	ERANGE
exp(x)	溢出	+Inf	ERANGE
exp(x)	下溢	+0	ERANGE
fmod(x,y)	x=Inf	QNaN	EDOM
fmod(x,y)	y = 0	QNaN	EDOM
log(x)	x < 0	QNaN	EDOM
log(x)	x = 0	-Inf	EDOM
log10(x)	x < 0	QNaN	EDOM
log10(x)	x = 0	-Inf	EDOM
pow(x,y)	溢出	+Inf	ERANGE
pow(x,y)	下溢	0	ERANGE
pow(x,y)	x=0 or x=Inf, y=0	+1	EDOM
pow(x,y)	x=+0, y<0	-Inf	EDOM
pow(x,y)	x=-0, y<0 and y integer	-Inf	EDOM
pow(x,y)	x= -0, y<0 and y non-integer	QNaN	EDOM
pow(x,y)	x<0, y non-integer	QNaN	EDOM
pow(x,y)	x=1, y=Inf	QNaN	EDOM

表 2-12 数学函数 （续）

敁悢	条件	返回值	错误编号
sqrt(x)	x < 0	QNaN	EDOM
sin(x)	x=Inf	QNaN	EDOM
sinh(x)	溢出	+Inf	ERANGE
tan(x)	x=Inf	QNaN	EDOM
atan(x)	SNaN	SNaN	无
ceil(x)	SNaN	SNaN	澇
floor(x)	SNaN	SNaN	澇
frexp(x)	SNaN	SNaN	澇
ldexp(x)	SNaN	SNaN	澇
modf(x)	SNaN	SNaN	澇
tanh(x)	SNaN	SNaN	澇

HUGE_VAL 是 Inf 的别名。有关错误编号，请参考 errno 变量。所有函数在传递 QNaN 时返回 QNaN，在传递 SNaN 时引发无效运算异常，但第2-96 页的表 2-12 中显示的情况除外。

将忽略传递给 C99 nan() 的字符串，并且始终返回相同的*非数字* (NaN)，即清除了最高位以外的所有小数位的 NaN。还会清除符号位。将 NAN(xxxx) 格式的字符串传递给 strtod 具有相同的效果。

信号函数

表 2-13 显示了 signal() 函数支持的信号。

表 2-13 信号函数

信号	编号	说明	附加自变量
SIGABRT	1	仅当应用程序调用 abort() 或 assert() 时，才会使用此信号。	澒
SIGFPE	2	用于发出任何算法异常信号，例如，除以零。由硬件浮点和软件浮点以及整数除法使用。	来自 {FE_EX_INEXACT, FE_EX_UNDERFLOW, FE_EX_OVERFLOW, FE_EX_DIVBYZERO, FE_EX_INVALID, DIVBYZERO} 的位集合
SIGILL	3	非法指令。	澒
SIGINT	4	来自用户的注意请求。	澒
SIGSEGV	5	内存访问错误。	澒
SIGTERM	6	终止请求。	澒
SIGSTAK	7	已不再使用。	澒
SIGRTRED	8	运行时库输入/输出流上的重定向失败。	为重定向标准流而重新打开的文件或设备的名称
SIGRTMEM	9	初始化期间或被破坏后出现堆空间不足。	失败请求的大小
SIGUSR1	10	由用户定义。	由用户定义
SIGUSR2	11	由用户定义。	由用户定义
SIGPVFN	12	从 C++ 中调用纯虚拟函数。	-
SIGCPPL	13	来自 C++ 的异常	-

表 2-13 信号函数 （续）

信号	编号	说明	附加自变量
SIGOUTOFHEAP	14	在出现堆空间不足时由 C++ 函数 ::operator new 返回。	失败请求的大小
保留	15-31	保留。	曇椒
其他	> 31	由用户定义。	由用户定义

虽然 **SIGSTAK** 存在于 signal.h 中，但 C 库不再生成此信号，此信号被视为已过时。

可通过 __raise() 传递编号大于 **SIGUSR2** 的信号，并由缺省信号处理程序捕获，但不能由使用 signal() 注册的处理程序捕获。

如果试图为编号大于 **SIGUSR2** 的信号注册处理程序， signal() 将返回一个错误代码。

所有可识别的信号的缺省处理方式是，输出诊断消息并调用 exit()。程序启动时将应用此缺省行为，直至您更改它时为止。

小心

适用于浮点处理的 IEEE 754 标准规定，对异常执行的缺省操作是继续运行而不进行捕获。缺省情况下，浮点计算中产生的异常不生成 **SIGFPE**。可通过调整 fenv.h 中的函数和定义来修改浮点错误处理方式。有关详细信息，请参阅 第2-58 页的 *调整错误信号*、*错误处理和程序退出*和第 4 章 *浮点支持*。

对于第2-98 页的表 2-13 中的所有信号，当发出信号时，如果处理程序指向一个函数，则在调用处理程序之前执行 signal(sig, SIG_DFL) 的等效函数。

如果 signal() 函数指定的处理程序收到 **SIGILL** 信号，则会重置缺省处理。

输入/输出特征

通用 ARM C 库具有以下输入/输出特征：

- 文本流的最后一行不需要终止换行符。
- 对于写出到文本流中紧靠换行符前面的空格字符，在重新读回后将会显示这些字符。

- 二进制输出流后面不附加空字符。
- 附加模式流的文件位置指示符最初放在文件末尾。
- 写入到文本流的操作导致在写入位置截断关联文件 （如果这是文件的设备类别行为）。
- 如果使用半主机，打开的最大文件数受可用目标内存限制。
- 存在零长度的文件，即，输出流没有在其中写入任何字符。
- 可以多次打开文件以进行读取，但写入或更新时只能打开一次。不能同时在一个流中打开某个文件以进行读取，而在另一个流中打开该文件以进行写入或更新。
- 未实现本地时区和夏令时。返回的值表示该信息不可用。例如， `gmtime()` 函数始终返回 `NULL`。
- `exit()` 返回的状态与传递给它的值相同。有关 `EXIT_SUCCESS` 和 `EXIT_FAILURE` 的定义，请参阅头文件 `stdlib.h`。但是，半主机并不将状态传回到执行环境。
- `strerror()` 函数返回的错误消息与 `perror()` 函数提供的消息完全相同。
- 如果请求的区域大小为零， `calloc()` 和 `realloc()` 将返回 `NULL`。
- 如果请求的区域大小为零， `malloc()` 将返回一个指向大小为零的块的指针。
- `abort()` 关闭所有打开的文件，并删除所有临时文件。
- `fprintf()` 以小写十六进制格式输出 `%p` 参数，就好像将精度指定为 8 一样。如果使用变体格式 (`%#p`)，则数字前面包含字符 `@`。
- `fscanf()` 将 `%p` 参数视为与 `%x` 参数完全相同。
- `fscanf()` 始终将 `%...[...]` 自变量中的字符-摄视为文字字符。
- `ftell()` 和 `fgetpos()` 在失败时将 `errno` 设置为 `EDOM` 的值。

- perror() 生成表 2-14 中显示的消息。

表 2-14 perror() 消息

错误	消息
0	No error (errno = 0)
EDOM	EDOM - function argument out of range
ERANGE	ERANGE - function result not representable
ESIGNUM	ESIGNUM - illegal signal number
其他	Unknown error

必须在符合 ISO 的实现中指定以下特征 （ARM C 库中没有指定这些特征）：

- 文件名的有效性
- remove() 能否删除打开的文件
- 在新名称已存在时调用 rename() 函数的效果
- 调用 getenv() 的效果 （缺省返回 NULL，无可用值）
- 调用 system() 的效果
- clock() 返回的值。

2.14.2 标准 C++ 库实现定义

本节介绍了 C++ 库实现。除了第2-102 页的表 2-15 中描述的某些限制外，ARM C++ 库提供了 *ISO/IEC 14822: 1998(E) C++ 标准* 中定义的所有库功能。

有关 Rogue Wave C++ 库中定义的实现定义行为的信息，请参阅随附的 Rogue Wave HTML 文档。缺省情况下，该文档安装在 *install_directory*\Documentation\RogueWave 中。

仅以二进制格式分发标准 C++ 库。

表 2-15 说明了当前版本中缺少的最重要功能。

表 2-15 标准 C++ 库差异

标准	实现差异
语言环境	不支持语言环境消息功能。无法在运行时打开目录，因为 ARM C 库不通过 `nl_types.h` 支持 `catopen` 和 `catclose`。可以在链接时选择两个语言环境定义之一。可通过用户重新定义的函数来创建其他语言环境。
时区	不支持。ARM C 库不支持时区。

2.15 C 库扩展

本节介绍了 C 库扩展和函数。有些扩展和函数是由 *ISO/IEC 9899:1999* C 标准定义的；有些扩展和函数因 ARM 编译器而异。表 2-16 简要介绍了这些函数和扩展。

表 2-16 C 库扩展

函数	头文件定义	扩展
atoll()	`stdlib.h`	C99 标准
第2-104 页的*strtoll()*	`stdlib.h`	C99 标准
第2-104 页的*strtoull()*	`stdlib.h`	C99 标准
第2-104 页的*printf()*	`stdlib.h`	C99 标准
第2-104 页的*snprintf()*	`stdio.h`	C99 标准
第2-105 页的*vsnprintf()*	`stdio.h`	C99 标准
第2-105 页的*lldiv()*	`stdlib.h`	C99 标准
第2-106 页的*llabs()*	`stdlib.h`	C99 标准
第2-106 页的*wcstombs()*	`stdlib.h`	POSIX 扩展功能
第2-106 页的*alloca()*	`alloca.h`	很多 C 库的通用非标准扩展
第2-107 页的*strlcpy()*	`string.h`	很多 C 库的通用 BSD 派生扩展
第2-107 页的*strlcat()*	`string.h`	很多 C 库的通用 BSD 派生扩展
第2-108 页的*_fisatty()*	`stdio.h`	因 ARM 编译器而异
第2-108 页的*__heapstats()*	`stdlib.h`	因 ARM 编译器而异
第2-109 页的*__heapvalid()*	`stdlib.h`	因 ARM 编译器而异

也可以使用 C99 中的头文件 `<stdint.h>` 和 `<inttypes.h>`。

2.15.1 atoll()

`atoll()` 函数将十进制字符串转换为整数。这与 ISO 函数 `atol()` 和 `atoi()` 类似，但返回 **`long long`** 结果。与 `atoi()` 一样，`atoll()` 可以接受八进制或十六进制输入（如果字符串以 `0` 或 `0x` 开头）。

语法

```
longlong atoll(const char *nptr)
```

2.15.2 strtoll()

strtoll() 函数将任意基址的字符串转换为整数。这与 ISO 函数 strtol() 类似，但返回 **long long** 结果。与 strtol() 一样， *endptr* 参数可保存指向转换后的字符串末尾的指针的存储位置，也可以是 NULL。*base* 参数必须包含数字基址。将 *base* 设置为 0 表示基址的选择方式与 atoll() 相同。

语法

```
longlong strtoll(const char *nptr, char **endptr, int base)
```

2.15.3 strtoull()

strtoull() 与 strtoll() 完全相同，但返回 **unsigned long long**。

语法

```
unsigned long long strtoull(const char *nptr, char **endptr, int base)
```

2.15.4 printf()

printf() 始终完全符合 ISO C89 标准。它还会有选择地支持 C99 中定义的其他格式指令，即，%a 和 %A（用于十六进制浮点）以及 %E、%F 和 %G（作为 %e、%f 和 %g 的大写版本）。缺省情况下，不包含这些 C99 格式指令。

要在 printf() 中启用 C99 功能，您必须指定 #pragma import(__use_c99_library)。这会影响 printf() 和 scanf() 系列中的所有函数。

语法

```
int printf(const char *format,  ...)
```

2.15.5 snprintf()

snprintf() 函数的工作方式几乎与 ISO sprintf() 函数完全相同，只不过调用方可以指定缓冲区的最大大小。返回值是设置了格式的完整字符串的长度，如果缓冲区足够大，则可能已写入该字符串。因此，仅当返回值最少为 0 且最多为 n-1 时，写入到缓冲区中的字符串才是完整的。

bufsize 参数指定函数可写入的 *buffer* 字符数，*包括* 终止空字符。

stdio.h 是 ISO 头文件。

ISO C90 标准禁止在 ISO 头文件中定义此函数。使用该函数通过 --c90 --strict 选项编译的代码将产生错误。

ISO C99 标准要求 ISO 头文件中包含此函数。使用 --c99 --strict 选项时，用此函数编译的代码不会产生错误。

语法

```
int snprintf(char *buffer, size_t bufsize, const char *format, ...)
```

2.15.6 vsnprintf()

vsnprintf() 函数的工作方式几乎与 ISO vsprintf() 函数完全相同，只不过调用方可以指定缓冲区的最大大小。返回值是设置了格式的完整字符串的长度，如果缓冲区足够大，则可能已写入该字符串。因此，仅当返回值最少为 0 且最多为 n-1 时，写入到缓冲区中的字符串才是完整的。

bufsize 参数指定函数可写入的 *buffer* 字符数，*包括* 终止空字符。

stdio.h 是 ISO 头文件。

ISO C90 标准禁止在 ISO 头文件中定义此函数。使用该函数通过 --c90 --strict 选项编译的代码将产生错误。

ISO C99 标准要求 ISO 头文件中包含此函数。使用该函数通过 --c99 --strict 选项编译的代码不会产生错误。

语法

```
int vsnprintf(char *buffer, size_t bufsize, const char *format, va_list ap)
```

2.15.7 lldiv()

lldiv 函数将两个 **long long** 整数相除并返回商和余数。它是 ISO 函数 ldiv 的 **long long** 等效函数。返回类型 lldiv_t 是包含两个 **long long** 成员（名为 quot 和 rem）的结构。

stdlib.h 是 ISO 头文件。

ISO C90 标准禁止在 ISO 头文件中定义此函数。使用该函数通过 --c90 --strict 选项编译的代码将产生错误。

ISO C99 标准要求 ISO 头文件中包含此函数。使用该函数通过 --c99 --strict 选项编译的代码不会产生错误。

语法

```
lldiv_t lldiv(long long num, long long denom)
```

2.15.8 llabs()

llabs() 函数返回其输入的绝对值。它是 ISO 函数 labs 的 **long long** 等效函数。

stdlib.h 是 ISO 头文件。

ISO C90 标准禁止在 ISO 头文件中定义此函数。使用该函数通过 --c90 --strict 选项编译的代码将产生错误。

ISO C99 标准要求 ISO 头文件中包含此函数。使用该函数通过 --c99 --strict 选项编译的代码不会产生错误。

语法

```
long long llabs(long long num)
```

2.15.9 wcstombs()

此函数按 ISO C 标准中所描述的方式工作，并具有 POSIX 所指定的扩展功能，即，如果 *s* 是空指针，wcstombs() 将返回转换整个数组所需的长度（而无论 *n* 的值是多少），但不会存储任何值。

语法

```
size_t wcstombs(char *s, const wchar_t *pwcs, size_t n)
```

2.15.10 alloca()

alloca() 函数在一个函数中分配本地存储。它返回一个指向内存 *size* 个字节的指针；如果没有足够的可用内存，则返回 NULL。缺省实现返回一个 8 字节对齐的内存块。

绝不能将从 alloca() 返回的内存传递给 free()。相反，当调用 alloca() 的函数返回时，将会自动释放内存。

不能通过函数指针调用 alloca()。在相同函数中使用 alloca() 和 setjmp() 时必须小心，因为 longjmp() 调用将释放 alloca() 在 setjmp() 和 longjmp() 调用之间分配的内存。

此函数是很多 C 库的通用非标准扩展。

语法

```
void* alloca(size_t size)
```

2.15.11 strlcpy()

strlcpy() 函数最多可将 *size*-1 个字符从以 NUL 终止的字符串 *src* 复制到 *dst*。它使用缓冲区的完整大小 （而不仅仅是长度），并且结果以 NUL 终止，但前提是 *size* 大于 0。将在 *size* 值中为 NUL 包含一个字节。

strlcpy() 函数返回*已*复制的字符串的总长度 （如果空间不受限制）。它可能等于*实际*复制的字符串长度，也可能不等于此长度，具体取决于是否有足够的空间。这意味着，可以调用一次 strlcpy() 以了解需要多少空间，然后分配空间（如果没有足够的空间），最后再次调用 strlcpy() 以进行所需的复制。

此函数是很多 C 库的通用 BSD 派生扩展。

语法

```
extern size_t strlcpy(char *dst, const char *src, size_t size)
```

2.15.12 strlcat()

strlcat() 函数连接两个字符串。它最多可将 *size*-strlen(*dst*)-1 个字节从以 NUL 终止的字符串 *src* 附加到 *dst* 末尾。它使用缓冲区的完整大小 （而不仅仅是长度），并且结果以 NUL 终止，但前提是 *size* 大于 0。将在 *size* 值中为 NUL 包含一个字节。

strlcat() 函数返回*已*创建的字符串的总长度 （如果空间不受限制）。它可能等于*实际*创建的字符串长度，也可能不等于此长度，具体取决于是否有足够的空间。这意味着，可以调用一次 strlcat() 以了解需要多少空间，然后分配空间（如果没有足够的空间），最后再次调用 strlcat() 以创建所需的字符串。

此函数是很多 C 库的通用 BSD 派生扩展。

语法

```
extern size_t strlcat(char *dst, *src, size_t size)
```

2.15.13 _fisatty()

_fisatty() 函数确定是否将给定的 stdio 流连接到终端设备或普通文件上。它在基本文件句柄上调用 _sys_istty() 低级函数。有关详细信息，请参阅第2-76 页的*调整输入/输出函数*。

此函数是 ARM 库特有的扩展。

语法

```
int _fisatty(FILE *stream)
```

返回值指示流目的地：

0 一个文件。

1 一个终端。

负数 错误。

2.15.14 __heapstats()

__heapstats() 函数显示有关内存分配堆状态的统计数据。ARM 编译器中的缺省实现提供有关存在的可用块数量的信息，并估计其大小范围。

示例 2-13 显示了一个来自 __heapstats() 的输出示例。

示例 2-13 来自 __heapstats() 的输出

```
32272 bytes in 2 free blocks (avge size 16136)
1 blocks 2^12+1 to 2^13
1 blocks 2^13+1 to 2^14
```

输出的第一行显示总字节数、可用块数和平均大小。后面几行显示每个块的估计大小 （按字节计算），并表示为一个范围。__heapstats() 并不提供有关已使用的块数的信息。

此函数通过调用输出函数 *dprint* 输出其结果，后者的工作方式必须与 fprintf() 类似。传递给 *dprint* 的第一个参数是提供的指针 *param*。可以传递 fprintf() 本身，但前提是将其转换为正确的函数指针类型。为了方便起见，将此类型定义为 **typedef**。它称为 __heapprt。例如：

```
__heapstats((__heapprt)fprintf, stderr);
```

注意

如果在尚未向其发送输出的流上调用 fprintf()，库将在内部调用 malloc()，以便为该流创建一个缓冲区。如果在调用 __heapstats() 的过程中发生这种情况，堆可能会毁坏。因此，必须确保已向 stderr 发送了一些输出。

如果使用缺省单区内存模型，则仅在需要时才分配堆内存。这意味着，在分配和释放内存时，可用堆的数量将会发生变化。例如，以下序列：

```
int *ip;
__heapstats((__heapprt)fprintf,stderr);   // print initial free heap size
ip = malloc(200000);
free(ip);
__heapstats((__heapprt)fprintf,stderr);    // print heap size after freeing
```

输出如下内容：

```
4076 bytes in 1 free blocks (avge size 4076)
1 blocks 2^10+1 to 2^11
2008180 bytes in 1 free blocks (avge size 2008180)
1 blocks 2^19+1 to 2^20
```

此函数是 ARM 库特有的扩展。

语法

```
void __heapstats(int (*dprint)( void *param, char const *format,...),
                 void* param)
```

2.15.15 __heapvalid()

__heapvalid() 函数对堆执行一致性检查。如果 *verbose* 参数为非零值，此函数将输出有关每个可用块的完整信息。否则，它只输出错误。

此函数通过调用输出函数 *dprint* 输出其结果，后者的工作方式必须与 fprintf() 类似。传递给 *dprint* 的第一个参数是提供的指针 *param*。可以传递 fprintf() 本身，但前提是将其转换为正确的函数指针类型。为了方便起见，将此类型定义为 **typedef**。它称为 __heapprt。例如：

示例 2-14 使用 fprintf() 调用 __heapvalid()

```
__heapvalid((__heapprt) fprintf, stderr, 0);
```

注意

如果在尚未向其发送输出的流上调用 fprintf()，库将在内部调用 malloc()，以便为该流创建一个缓冲区。如果在调用 __heapvalid() 的过程中发生这种情况，堆可能会毁坏。因此，必须确保已向 stderr 发送了一些输出。如果尚未在流中写入数据，示例 2-14 中的代码将会失败。

此函数是 ARM 库特有的扩展。

语法

```
int __heapvalid(int (*dprint)( void *param, char const *format,...), void*
                param, int verbose)
```

2.16 库命名约定

本节中介绍的库命名约定适用于 RVCT 的当前版本。不要依赖于库名称，因为它们在将来版本中可能会发生变化。

2.16.1 放置 ARM 库

通常，您不需要在链接器命令行中显式地列出其中的任何库。ARM 链接器根据累积的对象属性自动选择要使用的正确 C 或 C++ 库，并且可能会使用其中的一些库。

如果在 makefile 中显式地命名库名称，则必须按照以下方式重新构建项目：

1. 从链接器命令行中删除对旧库名称的显式引用。
2. 将 `--info libraries` 添加到链接器命令行中，然后重新构建项目。这将生成所使用的所有库的列表。
3. 将新的库列表添加到链接器命令行中。

要在分散加载描述文件中包含特定的库，请参阅 《链接器用户指南》 中第5-28 页的*放置 ARM 库*。

2.16.2 辅助函数

当编译器本身不能很容易地生成适合的代码序列时，通常将使用特定于 RVCT 的编译器支持，即*辅助*函数。

在 RVCT v4.0 及更高版本中，辅助函数由编译器在结果对象文件中生成。

在 RVCT v3.1 及更低版本中，辅助函数驻留在库中。由于这些库是 ARM C 编译器特有的，因此它们旨在根据需要与您自己的代码一起重新分发。例如，如果将库重新分发给第三方，可能还需要使用适当的辅助库才能成功链接最终应用程序。有关库的重新分发权限的详细信息，请参阅 《最终用户许可协议》。

2.16.3 标识库变体

库文件名指明了变体是如何构建的。文件名字段的值和相关构建选项如下所示：

`*root/prefix_arch[fpu][entrant].endian`

root
- `armlib` ARM C 库。
- `cpplib` ARM C++ 库。

prefix	c	ISO C 和 C++ 基本运行时支持。
	cpp	Rogue Wave C++ 库。
	cpprt	ARM C++ 运行时库。
	f	符合 IEEE 的库，具有固定舍入模式 （舍入到最接近的数）且没有不精确异常。
	fj	符合 IEEE 的库，具有固定舍入模式 （舍入到最接近的数）且没有异常。
	fz	其行为与 fj 库类似，但还会将非正规数和无限大刷新为 0。 该库的行为与快速模式下的 ARM VFP 类似。这是缺省设置。
	g	符合 IEEE 的库，具有可配置的舍入模式和所有 IEEE 异常。
	h	编译器支持 （辅助）库。请参阅第2-111 页的*辅助函数*。
	m	超数学函数。
	mc	不符合 ISO C 的微型库基本运行时支持。
	mf	不符合 IEEE 754 的微型库支持。
arch	4	与 ARMv4 配合使用且仅限 ARM 的库。
	t	与 ARMv4T 配合使用的 ARM/Thumb 交互操作库。
	5	与 ARMv5 及更高版本配合使用的 ARM/Thumb 交互操作库。
	w	与 Cortex-M3 配合使用且仅限 Thumb-2 的库。
	p	与 Cortex-M1 配合使用且仅限 Thumb-1 的库。
fpu	v	使用 VFP 指令集。
	s	软 VFP。 **——注意——** 如果库名称中既没有 v 也没有 s，则库不使用浮点。
entrant	e	与位置无关的静态数据访问。
	f	已启用 FPIC 寻址。

注意

如果库名称中既没有 e 也没有 f，则库使用与位置相关的静态数据访问。

endian	l	小端。
	b	大端。

例如：

```
*armlib/c_4.b
*cpplib/cpprt_5f.l
```

注意

并非所有变体/名称组合都有效。请查看 armlib 和 cpplib 目录以了解随 RVCT 提供的库。

链接器命令行选项 --info libraries 提供有关为链接阶段自动选择的每个库的信息。有关详细信息，请参阅 《链接器参考指南》中第2-25 页的*--info=topic[,topic,...]*。

有关详细信息，请参阅 《编译器用户指南》中第2-22 页的*指定目标处理器或体系结构*。

第 3 章
C 微型库

本章介绍 C *微型库* (microlib)。本章分为以下几节：

- 第3-2 页的*关于 microlib*
- 第3-4 页的*使用 microlib 构建应用程序*
- 第3-9 页的*调整 microlib 输入/输出函数*
- 第3-10 页的*microlib 中缺少的 ISO C 特性*

有关缺省库的详细信息，请参阅第 2 章 *C 和 C++ 库*。

3.1 关于 microlib

microlib 是缺省 C 库的备选库。它用于必须在极少量内存环境下运行的深层嵌入式应用程序。这些应用程序不在操作系统中运行。

注意

microlib 不会尝试成为符合标准的 ISO C 库。

microlib 进行了高度优化以使代码变得很小。它的功能比缺省 C 库少，并且根本不具备某些 ISO C 特性。某些库函数的运行速度也比较慢，例如，`memcpy()`。

3.1.1 与缺省 C 库之间的差异

microlib 与缺省 C 库之间的主要差异是：

- microlib 不符合 ISO C 库标准。不支持某些 ISO 特性，并且其他特性具有的功能也较少。
- microlib 不符合 IEEE 754 二进制浮点算法标准。
- microlib 进行了高度优化以使代码变得很小。
- 无法对语言环境进行配置。缺省 C 语言环境是唯一可用的语言环境。
- `main()` 不能声明为使用参数，并且不能返回内容。
- microlib 对 C99 函数提供有限的支持。
- microlib 不支持 C++。
- microlib 不支持操作系统函数。
- microlib 不支持与位置无关的代码。
- microlib 不提供互斥锁来防范非线程安全的代码。
- microlib 不支持宽字符或多字节字符串。
- 与标准库 (stdlib) 不同， microlib 不支持可选择的单区或双区内存模型。microlib 只提供双区内存模型，即单独的堆栈和堆区。
- microlib 可与 `--fpmode=std` 或 `--fpmode=fast` 一起使用。
- 提供的 ANSI C `stdio` 支持级别可通过 `#pragma import(__use_full_stdio)` 进行控制。

- `setvbuf()` 和 `setbuf()` 始终失败，因为所有流都未缓冲。
- `feof()` 和 `ferror()` 始终返回 `0`，因为不支持错误和 `EOF` 指示符。

有关详细信息，请参阅第3-10 页的*microlib 中缺少的 ISO C 特性*。

3.2 使用 microlib 构建应用程序

本节介绍如何将应用程序与 microlib 链接在一起。

microlib 中的函数负责：

- 创建一个可在其中执行 C 程序的环境。其中包括下列任务：
  - 创建一个堆栈
  - 创建一个堆 （如果需要）
  - 初始化程序所用的库的部分组成内容。
- 调用 `main()` 以开始执行程序。

要使用 microlib 构建程序，必须使用命令行选项 `--library_type=microlib`。此选项可由编译器、汇编器或链接器使用。此选项用于链接器时，会覆盖所有其他选项。

示例 3-1 显示了编译器对 `--library_type=microlib` 的使用。在编译 `main.c` 时指定 `--library_type=microlib` 会生成一个对象文件，其中包含一个要求链接器使用 microlib 的属性。不需要使用 `--library_type=microlib` 编译 `extra.c`，因为链接到 microlib 的请求位于通过编译 `main.c` 所生成的对象文件中。

示例 3-1 编译器选项

```
armcc --library_type=microlib -c main.c
armcc -c extra.c
armlink -o image.axf main.o extra.o
```

示例 3-2 显示了汇编器对此选项的使用。要求链接器使用 microlib 的请求是使用 `--library_type=microlib` 汇编 `more.s` 的结果。

示例 3-2 汇编器选项

```
armcc -c main.c
armcc -c extra.c
armasm --library_type=microlib more.s
armlink -o image.axf main.o extra.o more.o
```

第3-5 页的示例 3-3 显示了链接器对此选项的使用。任何对象文件都不包含要求链接器链接到 microlib 的属性，因此链接器选择 microlib 是在命令行显式要求这样做的结果。

示例 3-3 链接器选项

```
armcc -c main.c
armcc -c extra.c
armlink --library_type=microlib -o image.axf main.o extra.o
```

有关详细信息，请参阅：

- 《编译器参考指南》中第2-72 页的*--library_type=lib*
- 《链接器参考指南》中第2-29 页的*input_file_list*

3.3 使用 microlib

要开始使用，您必须为堆栈指定起始指针。要使用堆函数 （如 malloc()、calloc()、 realloc() 和 free()），您必须指定堆区的位置和大小。

3.3.1 创建堆栈

要指定初始堆栈指针，可以使用下列任一方法：

- 使用分散加载描述文件
- 定义一个等于堆栈顶部的符号 __initial_sp。

有关分散加载描述文件方法的信息，请参阅第2-71 页的*使用分散加载描述文件*中的 ARM_LIB_STACK 和 ARM_LIB_STACKHEAP。

另一种方法是，通过定义一个等于堆栈顶部的符号 __initial_sp 来指定初始堆栈指针。初始堆栈指针的对齐边界必须为 8 字节的倍数。

示例 3-4 说明了如何使用汇编语言来设置初始堆栈指针。

示例 3-4 汇编语言

```
    EXPORT __initial_sp
__initial_sp EQU 0x100000        ; equal to the top of the stack
```

示例 3-5 说明了如何使用 C 中的嵌入式汇编器来设置初始堆栈指针。

示例 3-5 C 中的嵌入式汇编器

```
__asm void dummy_function(void)
{
    EXPORT __initial_sp
__initial_sp EQU 0x100000        ; equal to the top of the stack
}
```

3.3.2 创建堆

要指定堆的开头和末尾，可以使用下列任一方法：

- 使用分散加载描述文件
- 定义符号 __heap_base 和 __heap_limit。

有关分散加载描述文件方法的信息，请参阅第2-71 页的*使用分散加载描述文件*中的 ARM_LIB_HEAP 和 ARM_LIB_STACKHEAP。

另一种方法是，通过分别定义符号 __heap_base 和 __heap_limit 来指定堆的开头和末尾。完成后，您可以按通常方式使用堆函数。

注意

__heap_limit 必须指向堆区中最后一个字节后面的字节。

示例 3-6 说明了如何使用汇编语言来设置堆指针。

示例 3-6 汇编语言

```
    EXPORT __heap_base
__heap_base EQU 0x400000        ; equal to the start of the heap
    EXPORT __heap_limit
__heap_limit EQU 0x800000       ; equal to the end of the heap
```

示例 3-7 说明了如何使用 C 中的嵌入式汇编器来设置堆指针。

示例 3-7 C 中的嵌入式汇编器

```
__asm void dummy_function(void)
{
    EXPORT __heap_base
__heap_base EQU 0x400000        ; equal to the start of the heap
    EXPORT __heap_limit
__heap_limit EQU 0x800000       ; equal to the end of the heap
}
```

3.3.3 进入和退出程序

使用 main() 开始您的程序。不要将 main() 声明为使用参数。

注意

程序不能从 main() 返回内容。

microlib 不支持以下内容：

- 操作系统中的命令行参数
- 调用 exit() 的程序。

3.4 调整 microlib 输入/输出函数

microlib 提供了一个有限的 stdio 子系统，它仅支持未缓冲的 stdin、stdout 和 stderr。这样，即可使用 printf() 来显示应用程序中的诊断消息。

要使用高级 I/O 函数，您必须提供自己实现的下列基本函数，以便与您自己的 I/O 设备配合使用。

fputc()　为所有输出函数实现此基本函数。例如，fprintf()、printf()、fwrite()、fputs()、puts()、putc() 和 putchar()。

fgetc()　为所有输入函数实现此基本函数。例如，fscanf()、scanf()、fread()、read()、fgets()、gets()、getc() 和 getchar()。

__backspace()

如果输入函数使用 scanf() 或 fscanf()，则实现此基本函数。

注意

microlib 中不支持的转换为 %lc、%ls 和 %a。

有关详细信息，请参阅第2-76 页的*调整输入/输出函数*。

3.5 microlib 中缺少的 ISO C 特性

本节提供了 microlib 不支持的主要 ISO C90 特性的列表。

宽字符和多字节支持

microlib 不支持所有处理宽字符或多字节字符串的函数。如果使用这些函数，则会产生链接器错误。例如，`mbtowc()`、`wctomb()`、`mbstowcs()` 和 `wcstombs()`。microlib 不支持在标准附录 1 中定义的所有函数。

操作系统交互

microlib 不支持与操作系统交互的所有函数。例如 `abort()`、`exit()`、`atexit()`、`clock()`、`time()`、`system()` 和 `getenv()`。

文件 I/O 缺省情况下，与文件指针交互的所有 `stdio` 函数都将返回错误 （如果调用）。仅有的例外情况是以下三个标准流：`stdin`、`stdout` 和 `stderr`。

可以使用 `#pragma import(__use_full_stdio)` 更改此行为。使用此编译指示可提供支持 ANSI C 的 `stdio` 的 microlib 版本，只有以下几种例外情况：

- 不支持错误和 `EOF` 指示符，因此 `feof()` 和 `ferror()` 返回 `0`。
- 所有流都未缓冲，因此 `setvbuf()` 和 `setbuf()` 会失败。

可配置的语言环境

缺省 C 语言环境是唯一可用的语言环境。

信号 虽然提供了 `signal()` 和 `raise()` 函数，但 microlib 不会生成信号。唯一的例外情况是程序显式地调用 `raise()`。

浮点支持 浮点支持在以下几个方面不符合 IEEE 754，但在仅涉及规范化数的运算中使用相同的数据格式并符合 IEEE 754：

- 涉及 NaN、无穷大或非正规数的运算可能会产生不可预测的结果。
- microlib 不能标记 IEEE 异常，并且 microlib 中没有 `fp_status()` 寄存器。
- microlib 不会将零的符号视为有效位，并且 microlib 浮点算法输出中的零可能会包含不可预测的符号位。
- 仅支持缺省的舍入模式。

与位置无关且线程安全的代码

microlib 没有可重入变体。microlib 不提供互斥锁来防范非线程安全的代码。microlib 的使用与 FPIC 或 RWPI 编译模式不兼容，但可以将 ROPI 代码与 microlib 进行链接，生成的二进制文件总体上与 ROPI 不兼容。

第 4 章
浮点支持

本章介绍了 ARM 对浮点计算的支持。本章分为以下几节：

- 第4-2 页的*软件浮点库 fplib*
- 第4-10 页的*控制浮点环境*
- 第4-26 页的*数学库 mathlib*
- 第4-35 页的*IEEE 754 算法*

4.1 软件浮点库 fplib

当为使用浮点协处理器而编译程序时，程序通过目标协处理器的浮点机器指令来执行基本浮点算法。但是，当为使用软件浮点而编译程序时，则没有浮点指令集可用，因此， ARM 库必须提供一组过程调用来执行浮点算法。这些过程位于软件浮点库 fplib 中。

4.1.1 浮点库 fplib 的特性

浮点例程具有类似于 `__aeabi_dadd` （将两个 **double** 相加）和 `__aeabi_fdiv` （将两个 **float** 相除）的名称。用户程序可直接调用这些例程。即使在使用协处理器的环境中，也会提供这些例程。不过，这些例程通常只有几个指令长 （因为它们仅限执行相应的协处理器指令）。

所有 fplib 例程都是使用调用标准的软件浮点变体调用的。这意味着，在整数寄存器中传递并返回浮点参数。与此相反，如果为协处理器编译程序，则会在浮点寄存器中传递浮点数据。

因此，举例来说，`__aeabi_dadd` 在 `r0` 和 `r1` 寄存器中提取 **double**，在 `r2` 和 `r3` 寄存器中提取另一个 **double**，并在 `r0` 和 `r1` 中返回两者的和。

注意

对于 `r0` 和 `r1` 寄存器中的 **double**，保存 **double** 的高 32 位的寄存器取决于程序是小端还是大端。

所有 fplib 例程（第4-8 页的表 4-5 中列出的例程除外）都是在头文件 `rt_fp.h` 中声明的。如果要直接调用 fplib 例程，则可以包含此文件。第4-8 页的表 4-5 中显示的例程是在标准头文件 `math.h` 中声明的。

要从汇编器中调用函数，请按 `__softfp_`*fn* 格式调用软件浮点函数。例如，要调用 `cos()` 函数，请执行以下命令：

```
IMPORT __softfp_cos
BL __softfp_cos
```

4.1.2 特定格式数字的算法

表 4-1 介绍了对特定格式的数字执行算法的例程。参数和结果始终采用相同的格式。

表 4-1 算法例程

函数	自变量类型	结果类型	运算
__aeabi_fadd	2 **float**	**float**	返回 x 加 y
__aeabi_fsub	2 **float**	**float**	返回 x 减 y
__aeabi_frsub	2 **float**	**float**	返回 y 减 x
__aeabi_fmul	2 **float**	**float**	返回 x 乘以 y
__aeabi_fdiv	2 **float**	**float**	返回 x 除以 y
_frdiv	2 **float**	**float**	返回 y 除以 x
_frem	2 **float**	**float**	返回 x 除以 y 的余数（请参阅第4-4 页的*算法例程注释*中的 a）
_frnd	**float**	**float**	返回 x 取整后的值 （请参阅第4-4 页的*算法例程注释*中的 b）
_fsqrt	**float**	**float**	返回 x 的平方根
__aeabi_dadd	2 **double**	**double**	返回 x 加 y
__aeabi_dsub	2 **double**	**double**	返回 x 减 y
__aeabi_drsub	2 **double**	**double**	返回 y 减 x
__aeabi_dmul	2 **double**	**double**	返回 x 乘以 y
__aeabi_ddiv	2 **double**	**double**	返回 x 除以 y
_drdiv	2 **double**	**double**	返回 y 除以 x
_drem	2 **double**	**double**	返回 x 除以 y 的余数（请参阅第4-4 页的*算法例程注释*中的 a）
_drnd	**double**	**double**	返回 x 取整后的值 （请参阅第4-4 页的*算法例程注释*中的 b）
_dsqrt	**double**	**double**	返回 x 的平方根

算法例程注释

a 执行 IEEE 754 求余数运算的函数。此函数被定义为使用两个数 *x* 和 *y*，并返回一个数 *z*，以使 $z = x - n * y$，其中 *n* 为整数。要返回完全正确的结果，请选择 *n* 以使 *z* 不大于 *x* 的一半（因此，即使 *x* 和 *y* 都是正数， *z* 也可能为负数）。IEEE 754 求余数函数与 C 库函数 `fmod` 执行的运算不同，其中 *z* 的符号始终与 *x* 相同。如果 IEEE 754 规范给出了两个可接受的 *n* 选项，则选择偶数选项。出现的此行为与当前舍入模式无关。

b 执行 IEEE 754 取整运算的函数。此函数使用一个数并取整（根据当前舍入模式），但以浮点数格式返回该整数，而不是作为 C **int** 变量。要将一个数转换为 **int** 变量，必须使用表 4-2 中介绍的 `_ffix` 例程。

4.1.3 浮点数、双精度数和整数之间的转换

表 4-2 介绍了在数字格式之间执行转换的例程（**long long** 除外）。

表 4-2 数字格式转换例程

函数	自变量类型	结果类型
`__aeabi_f2d`	**float**	**double**
`__aeabi_d2f`	**double**	**float**
`_fflt`	**int**	**float**
`_ffltu`	**unsigned int**	**float**
`_dflt`	**int**	**double**
`_dfltu`	**unsigned int**	**double**
`_ffix`	**float**	**int**（请参阅第4-5 页的*舍入注释*）
`_ffix_r`	**float**	**int**
`_ffixu`	**float**	**unsigned int**（请参阅第4-5 页的*舍入注释*）
`_ffixu_r`	**float**	**unsigned int**
`_dfix`	**double**	**int**（请参阅第4-5 页的*舍入注释*）

表 4-2 数字格式转换例程 （续）

函数	自变量类型	结果类型
_dfix_r	**double**	**int**
_dfixu	**double**	**unsigned int** （请参阅*舍入注释*）
_dfixu_r	**double**	**unsigned int**

舍入注释

向零舍入与当前舍入模式无关。这是因为， C 标准要求隐式转换为以这种方式舍入的整数，由于不需要更改舍入模式，因而操作起来非常方便。每个函数都有名称以 _r 结尾的对应函数；对应函数执行相同的运算，但根据当前模式进行舍入。

4.1.4 long long 和其他数字格式之间的转换

表 4-3 介绍了在 **long long** 和其他数字格式之间执行转换的例程。

表 4-3 涉及 long long 格式的转换例程

函数	自变量类型	结果类型
_ll_sto_f	**long long**	**float**
_ll_uto_f	**无符号 long long**	**float**
_ll_sto_d	**long long**	**double**
_ll_uto_d	**无符号 long long**	**double**
_ll_sfrom_f	**float**	**long long** （请参阅第4-6 页的*舍入注释*）
_ll_sfrom_f_r	**float**	**long long**
_ll_ufrom_f	**float**	**无符号 long long** （请参阅第4-6 页的*舍入注释*）
_ll_ufrom_f_r	**float**	**无符号 long long**
_ll_sfrom_d	**double**	**long long** （请参阅第4-6 页的*舍入注释*）

表 4-3 涉及 long long 格式的转换例程 （续）

函数	自变量类型	结果类型
_ll_sfrom_d_r	**double**	**long long**
_ll_ufrom_d	**double**	**无符号 long long**（请参阅*舍入注释*）
_ll_ufrom_d_r	**double**	**无符号 long long**

舍入注释

向零舍入与当前舍入模式无关。这是因为， C 标准要求隐式转换为以这种方式舍入的整数，由于不需要更改舍入模式，因而操作起来非常方便。此函数具有名称以 _r 结尾的对应函数。对应函数执行相同的运算，但根据当前模式进行舍入。

4.1.5 浮点数比较

表 4-4 介绍了在浮点数之间执行比较的例程。有关详细信息，请参阅第4-7 页的*浮点数比较例程注释*中的**注释**列。

表 4-4 浮点数比较例程

函数	自变量类型	结果类型	测试的条件	说明
_fcmpeq	2 **float**	标记， EQ/NE	x 等于 y	a
_fcmpge	2 **float**	标记， HS/LO	x 大于或等于 y	a、 b
_fcmple	2 **float**	标记， HI/LS	x 小于或等于 y	a、 b
_feq	2 **float**	布尔值	x 等于 y	-
_fneq	2 **float**	布尔值	x 不等于 y	-
_fgeq	2 **float**	布尔值	x 大于或等于 y	b
_fgr	2 **float**	布尔值	x 大于 y	b
_fleq	2 **float**	布尔值	x 小于或等于 y	b
_fls	2 **float**	布尔值	x 小于 y	b
_dcmpeq	2 **double**	标记， EQ/NE	x 等于 y	a

表 4-4 浮点数比较例程 （续）

函数	自变量类型	结果类型	测试的条件	说明
_dcmpge	2 **double**	标记， HS/LO	x 大于或等于 y	a、 b
_dcmple	2 **double**	标记， HI/LS	x 小于或等于 y	a、 b
_deq	2 **double**	布尔值	x 等于 y	-
_dneq	2 **double**	布尔值	x 不等于 y	-
_dgeq	2 **double**	布尔值	x 大于或等于 y	b
_dgr	2 **double**	布尔值	x 大于 y	b
_dleq	2 **double**	布尔值	x 小于或等于 y	b
_dls	2 **double**	布尔值	x 小于 y	b
_fcmp4	2 **float**	标记， VFP	x 小于或等于 y	c
_fcmp4e	2 **float**	标记， VFP	x 小于或等于 y	b、 c
_fdcmp4	**float**、 **double**	标记， VFP	x 小于或等于 y	c
_fdcmp4e	**float**、 **double**	标记， VFP	x 小于或等于 y	b、 c
_dcmp4	2 **double**	标记， VFP	x 小于或等于 y	c
_dcmp4e	2 **double**	标记， VFP	x 小于或等于 y	b、 c
_dfcmp4	**double**、 **float**	标记， VFP	x 小于或等于 y	c
_dfcmp4e	**double**、 **float**	标记， VFP	x 小于或等于 y	b、 c

浮点数比较例程注释

a 在 ARM 条件标记中返回结果。这在汇编语言中是非常有效的，因为您可以直接在函数调用后面执行条件指令，但这表示无法从 C 中使用此函数。此函数未在 rt_fp.h 中声明。

b 如果任一自变量是 NaN（即使是无提示 NaN），则会导致“无效运算”异常。仅当自变量是SNaN时，其他函数才会导致“无效运算”。与任意数 （包括其他 QNaN）进行比较时， QNaN 返回*不等于*（因此，将 QNaN 与相同的 QNaN 比较时，仍会返回*不等于*）。

c 在 CPSR 中返回 VFP 类型状态标记。还在 r0 的最高四位中返回 VFP 类型状态标记，这表示可以从 C 中使用此函数。此函数是在 rt_fp.h 中声明的。

4.1.6 C99 函数

表 4-5 介绍了实现 C99 功能的例程。

表 4-5 C99 例程

函数	自变量类型	结果类型	返回节	标准
ilogb	**double**	**int**	自变量 x 的指数	7.12.6.5
ilogbf	**float**	**int**	自变量 x 的指数	7.12.6.5
ilogbl	**long double**	**int**	自变量 x 的指数	7.12.6.5
logb	**double**	**double**	自变量 x 的指数	7.12.6.11
logbf	**float**	**float**	自变量 x 的指数	7.12.6.11
logbl	**long double**	**long double**	自变量 x 的指数	7.12.6.11
scalbn	**double**、 **int**	**double**	x * (FLT_RADIX ** n)	7.12.6.13
scalbnf	**float**、 **int**	**float**	x * (FLT_RADIX ** n)	7.12.6.13
scalbnl	**long double**、 **int**	**long double**	x * (FLT_RADIX ** n)	7.12.6.13
scalbln	**double**、 **long int**	**double**	x * (FLT_RADIX ** n)	7.12.6.13
scalblnf	**float**、 **long int**	**float**	x * (FLT_RADIX ** n)	7.12.6.13
scalblnl	**long double**、 **long int**	**long double**	x * (FLT_RADIX ** n)	7.12.6.13
nextafter	2 **double**	**double**	在指向 y 的方向， x 之后下一个可表示的值	7.12.11.3
nextafterf	2 **float**	**float**	在指向 y 的方向， x 之后下一个可表示的值	7.12.11.3
nextafterl	2 **long double**	**long double**	在指向 y 的方向， x 之后下一个可表示的值	7.12.11.3

表 4-5 C99 例程 （续）

函数	自变量类型	结果类型	返回节	标准
nexttoward	**double**、**long double**	**double**	在指向 y 的方向， x 之后下一个可表示的值	7.12.11.4
nexttowardf	**float**、**long double**	**float**	在指向 y 的方向， x 之后下一个可表示的值	7.12.11.4
nexttowardl	2 **long double**	**long double**	在指向 y 的方向， x 之后下一个可表示的值	7.12.11.4

4.2 控制浮点环境

本节介绍了可用于控制 ARM 浮点环境的函数。RVCT 为浮点环境提供了几个不同接口，以保持兼容和便于移植。通过使用这些函数，可以更改舍入模式、启用和禁用异常捕获以及安装您自己的自定义异常捕获处理程序。

4.2.1 __ieee_status()

RVCT 在浮点环境中支持状态字的接口。它称为 __ieee_status，通常是用于修改 VFP 状态字的最有效函数。__ieee_status 是在 fenv.h 中定义的。

__ieee_status 具有以下原型：

```
unsigned int __ieee_status(unsigned int mask, unsigned int flags);
```

该函数根据参数修改状态字的可写部分，并返回整个字先前的值。

可写位的修改方式是，将它们设置为：

```
new = (old & ~mask) ^ flags;
```

可以对状态字的每个位执行四种不同的运算，具体取决于掩码和标记中的对应位（请参阅表 4-6）。

表 4-6 状态字位修改

掩码位	标记位	效果
0	0	保持不变
0	1	切换
1	0	设置为 0
1	1	设置为 1

图 4-1 中给出了 __ieee_status 所显示的状态字布局。

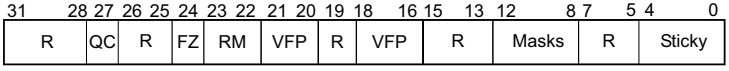

图 4-1 IEEE 状态字布局

第4-10 页的图 4-1 中的字段如下所示：

- 第 0 位到第 4 位 （值分别为 0x1 到 0x10）是每个异常的粘性标记或累积标记。只要发生异常而未将其捕获，就会将该异常的粘性标记设置为 1。系统从不清除粘性标记，而只能由用户将其清除。异常与位之间的映射是：
  - 第 0 位 (0x01) 用于 “无效运算”异常
  - 第 1 位 (0x02) 用于 “除以零”异常
  - 第 2 位 (0x04) 用于 “溢出”异常
  - 第 3 位 (0x08) 用于 “下溢”异常
  - 第 4 位 (0x10) 用于 “不精确结果”异常
- 第 8 位到第 12 位 （值为 0x100 到 0x1000）是异常掩码。这些位控制是否捕获每个异常。如果将某个位设置为 1，则会捕获相应的异常。如果将某个位设置为 0，相应异常将设置其粘性标记，并返回一个似是而非的结果，如第4-39 页的*异常*中所述。
- VFP 硬件使用第 16 位到第 18 位、第 20 位以及第 21 位来控制 VFP 向量功能。__ieee_status 调用不允许修改这些位。
- 第 22 位和第 23 位控制舍入模式 （表 4-7）。

表 4-7 舍入模式控制

位	舍入模式
00	舍入到最接近的数
01	向上舍入
10	向下舍入
11	向零舍入

注意

标准 fplib 库 f* 仅支持“舍入到最接近的数”舍入模式。如果需要其他舍入模式支持，必须使用完整的 IEEE g* 库。请参阅第2-111 页的*库命名约定*。

- 如果设置了第 24 位，则会启用 FZ （清零）模式。在 FZ 模式下，强制将非正规数刷新为零以加快处理速度 （因为非正规数可能很难与浮点系统一起使用，并且会降低浮点系统的速度）。设置此位会降低准确性，但可能会提高速度。

注意

标准 fplib 库不支持 FZ 模式。相反，每个库要么始终刷新为零，要么从不刷新为零，并且您可以选择在构建时使用的库。仅当设置了第 24 位时，VFP 硬件才会支持 FZ 模式。

但是，这意味着硬件中没有提供的函数 （因而在软件 fplib 中提供）不支持 FZ 模式，即使为硬件 VFP 进行编译时也是如此。因此，在动态更改 FZ 模式时，所有函数中的 fplib 行为并不一致。

- 第 27 位指示高级 SIMD 饱和整数运算中已发生饱和。可通过 __ieee_status 调用来访问该位。
- 标记为 R 的位是保留位。__ieee_status 调用无法写入这些位，必须忽略其中包含的任何信息。

除了定义 __ieee_status 调用本身之外， fenv.h 还定义了一些用于参数的常数：

```
#define FE_IEEE_FLUSHZERO          (0x01000000)
#define FE_IEEE_ROUND_TONEAREST    (0x00000000)
#define FE_IEEE_ROUND_UPWARD       (0x00400000)
#define FE_IEEE_ROUND_DOWNWARD     (0x00800000)
#define FE_IEEE_ROUND_TOWARDZERO   (0x00C00000)
#define FE_IEEE_ROUND_MASK         (0x00C00000)
#define FE_IEEE_MASK_INVALID       (0x00000100)
#define FE_IEEE_MASK_DIVBYZERO     (0x00000200)
#define FE_IEEE_MASK_OVERFLOW      (0x00000400)
#define FE_IEEE_MASK_UNDERFLOW     (0x00000800)
#define FE_IEEE_MASK_INEXACT       (0x00001000)
#define FE_IEEE_MASK_ALL_EXCEPT    (0x00001F00)
#define FE_IEEE_INVALID            (0x00000001)
#define FE_IEEE_DIVBYZERO          (0x00000002)
#define FE_IEEE_OVERFLOW           (0x00000004)
#define FE_IEEE_UNDERFLOW          (0x00000008)
#define FE_IEEE_INEXACT            (0x00000010)
#define FE_IEEE_ALL_EXCEPT         (0x0000001F)
```

例如，要将舍入模式设置为向下舍入，应执行以下命令：

```
__ieee_status(FE_IEEE_ROUND_MASK, FE_IEEE_ROUND_DOWNWARD);
```

要捕获 “无效运算”异常而不捕获所有其他异常，应执行以下命令：

```
__ieee_status(FE_IEEE_MASK_ALL_EXCEPT, FE_IEEE_MASK_INVALID);
```

要不捕获 “不精确结果”异常，应执行以下命令：

```
__ieee_status(FE_IEEE_MASK_INEXACT, 0);
```

要清除 “下溢”粘性标记，应执行以下命令：

```
__ieee_status(FE_IEEE_UNDERFLOW, 0);
```

4.2.2 __fp_status()

以前版本的 ARM 库实现了一个名为 __fp_status 的函数，用于在浮点环境中处理状态字。它与 __ieee_status 相同，但使用的是旧式状态字布局。ARM 编译器仍然支持 __fp_status 函数以保持向后兼容。__fp_status 是在 stdlib.h 中定义的。

__fp_status 具有以下原型：

```
unsigned int __fp_status(unsigned int mask, unsigned int flags);
```

图 4-2 中给出了 __fp_status 所显示的状态字布局。

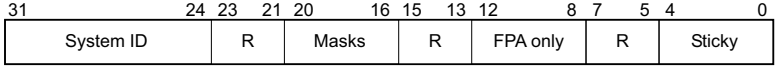

图 4-2 浮点状态字布局

图 4-2 中的字段如下：

- 第 0 位到第 4 位是粘性标记，如第4-10 页的*__ieee_status()* 中所述。
- 第 8 位到第 12 位（值为 0x100 到 0x1000）控制 FPA 浮点的各个方面。FPA 体系结构已过时， RVCT 不再提供支持。将忽略任何试图写入这些位的操作。
- 第 16 位到 20 位 （值为 0x10000 到 0x100000）是异常掩码 （如第4-10 页的 *__ieee_status()* 中所述），但位于不同的位置。
- 第 24 位到第 31 位包含无法更改的系统 ID。对于软件浮点，它被设置为 0x40；对于硬件浮点，它被设置为 0x80 或更大的值；如果仿真器仿造硬件浮点环境，则设置为 0 或 1。
- 标记为 R 的位是保留位。__fp_status 调用无法写入这些位，必须忽略其中包含的任何信息。

无法使用 __fp_status 调用更改舍入模式。

除了定义 __fp_status 调用本身之外， stdlib.h 还定义了一些用于参数的常数：

```
#define __fpsr_IXE  0x100000
#define __fpsr_UFE  0x80000
#define __fpsr_OFE  0x40000
#define __fpsr_DZE  0x20000
#define __fpsr_IOE  0x10000
#define __fpsr_IXC  0x10
#define __fpsr_UFC  0x8
#define __fpsr_OFC  0x4
#define __fpsr_DZC  0x2
#define __fpsr_IOC  0x1
```

例如，要捕获 “无效运算”异常而不捕获所有其他异常，应执行以下命令：

```
__fp_status(_fpsr_IXE | _fpsr_UFE | _fpsr_OFE |
            _fpsr_DZE | _fpsr_IOE, _fpsr_IOE);
```

要不捕获 “不精确结果”异常，应执行以下命令：

```
__fp_status(_fpsr_IXE, 0);
```

要清除 “下溢”粘性标记，应执行以下命令：

```
__fp_status(_fpsr_UFC, 0);
```

4.2.3 Microsoft 兼容性函数

为了与 Microsoft 产品保持兼容，实现了以下函数，以便于将浮点代码移植到 ARM 体系结构中。这些函数是在 `float.h` 中定义的。

_controlfp()

可以使用 `_controlfp()` 函数来控制异常捕获和舍入模式：

```
unsigned int _controlfp(unsigned int new, unsigned int mask);
```

此函数还通过使用掩码隔离要修改的位来修改控制字。对于每个为零的 `mask` 位，相应控制字位保持不变。对于每个非零的 `mask` 位，相应控制字位被设置为对应 `new` 位的值。返回值是以前的控制字状态。

注意

它与 `__ieee_status()` （或 `__fp_status()`）的行为并不完全相同；对于后者，可通过在掩码字中设置 0 并在标记字中设置 1 对位进行切换。

表 4-8 介绍了可用于构成 _controlfp() 参数的宏。

表 4-8 _controlfp 自变量宏

宏	说明
_MCW_EM	包含所有异常位的掩码
_EM_INVALID	描述 “无效运算”异常的位
_EM_ZERODIVIDE	描述 “除以零”异常的位
_EM_OVERFLOW	描述 “溢出”异常的位
_EM_UNDERFLOW	描述 “下溢”异常的位
_EM_INEXACT	描述 “不精确结果”异常的位
_MCW_RC	舍入模式字段的掩码
_RC_CHOP	描述 “向零舍入”的舍入模式值
_RC_UP	描述 “向上舍入”的舍入模式值
_RC_DOWN	描述 “向下舍入”的舍入模式值
_RC_NEAR	描述 “舍入到最接近的数”的舍入模式值

注意

无法保证这些宏的值在将来版本的 ARM 产品中保持不变。要确保这些值在将来版本中发生变化时代码仍然有效，请使用宏而不是使用其值。

例如，要将舍入模式设置为向下舍入，应执行以下命令：

```
_controlfp(_RC_DOWN, _MCW_RC);
```

要捕获 “无效运算”异常而不捕获所有其他异常，应执行以下命令：

```
_controlfp(_EM_INVALID, _MCW_EM);
```

要不捕获 “不精确结果”异常，应执行以下命令：

```
_controlfp(0, _EM_INEXACT);
```

_clearfp()

_clearfp() 函数清除所有五个异常粘性标记并返回其先前的值。第4-15 页的表 4-8 中给出的宏 （如 _EM_INVALID、 _EM_ZERODIVIDE）可用于测试返回结果的位。

_clearfp() 具有以下原型：

```
unsigned _clearfp(void);
```

_statusfp()

_statusfp() 函数返回异常粘性标记的当前值。可以使用第4-15 页的表 4-8 中显示的宏来测试返回结果的位，例如 _EM_INVALID、 _EM_ZERODIVIDE。

_statusfp 具有以下原型：

```
unsigned _statusfp(void);
```

4.2.4 C99 兼容函数

除了前面介绍的函数以外， ARM 编译器还支持在 C99 标准中定义的一组函数。

这些 C99 兼容函数是可用于安装自定义异常捕获处理程序 （具有生成返回值的功能）的唯一接口。本节中的所有函数、类型和宏都是在 fenv.h 中定义的。

C99 定义了以下两种数据类型：fenv_t 和 fexcept_t。C99 标准未提供有关这些类型的任何信息，因此，对于可移植代码，必须将这些类型视为不透明的类型。ARM 编译器将它们定义为结构类型。有关详细信息，请参阅第4-19 页的*ARM 编译器的 C99 接口扩展*。

fenv_t 类型被定义为存储有关当前浮点环境的所有信息：

- 舍入模式
- 异常粘性标记
- 是否掩盖了每个异常
- 安装的处理程序 （如果有）。

fexcept_t 类型被定义为存储与一组给定异常相关的所有信息。

C99 舍入模式和异常宏

C99 还为每种舍入模式和每个异常定义了一个宏。这些宏如下所示：

```
FE_DIVBYZERO
FE_INEXACT
FE_INVALID
FE_OVERFLOW
FE_UNDERFLOW
FE_ALL_EXCEPT
FE_DOWNWARD
FE_TONEAREST
FE_TOWARDZERO
FE_UPWARD
```

异常宏是位域。FE_ALL_EXCEPT 宏是所有这些字段的按位 “或” 运算。

处理异常标记

C99 提供了以下三个用于清除、测试和产生异常的函数：

```
void feclearexcept(int excepts);
int fetestexcept(int excepts);
void feraiseexcept(int excepts);
```

feclearexcept() 函数清除给定异常的粘性标记。fetestexcept() 函数返回给定异常的粘性标记的按位“或”运算结果（因此，如果设置了“溢出”标记而未设置“下溢”标记， fetestexcept(FE_OVERFLOW|FE_UNDERFLOW) 调用将返回 FE_OVERFLOW）。

feraiseexcept() 函数按未指定的顺序产生给定的异常。如果为以这种方式产生的异常启用了异常捕获，则会调用该函数。

C99 还提供了一些函数以保存和恢复与给定异常有关的所有内容。这包括粘性标记、是否捕获异常以及捕获处理程序地址 （如果有）。这些函数是：

```
void fegetexceptflag(fexcept_t *flagp, int excepts);
void fesetexceptflag(const fexcept_t *flagp, int excepts);
```

fegetexceptflag() 函数将与给定异常有关的所有信息复制到提供的 fexcept_t 变量中。fesetexceptflag() 函数将与给定异常有关的所有信息从 fexcept_t 变量复制到当前浮点环境中。

——注意——

可以使用 fesetexceptflag() 将捕获的异常的粘性标记设置为 1，而无需调用捕获处理程序，但 feraiseexcept() 会为所有捕获的异常调用捕获处理程序。

处理舍入模式

C99 提供了以下两个用于控制舍入模式的函数：

```
int fegetround(void);
int fesetround(int round);
```

fegetround() 函数返回当前舍入模式，该模式的值等于第4-17 页的*C99 舍入模式和异常宏*中列出的一个宏的值。fesetround() 函数将当前舍入模式设置为提供的值。如果成功， fesetround() 将返回零；如果其自变量不是有效舍入模式，则返回非零值。

保存整个环境

C99 提供了一些用于保存和恢复整个浮点环境的函数：

```
void fegetenv(fenv_t *envp);
void fesetenv(const fenv_t *envp);
```

fegetenv() 函数将浮点环境的当前状态存储到提供的 fenv_t 变量中。fesetenv() 函数从提供的变量中恢复环境。

与 fesetexceptflag() 一样， fesetenv() 在为捕获的异常设置粘性标记时不调用捕获处理程序。

临时禁用异常

C99 提供了两个具有以下功能的函数：规避了在执行可能导致出现异常的代码时进行异常捕获的风险。这是非常有用的，例如，在捕获的异常使用 ARM 缺省行为时。缺省行为是导致生成 **SIGFPE** 并终止应用程序。

```
int feholdexcept(fenv_t *envp);
void feupdateenv(const fenv_t *envp);
```

feholdexcept() 函数将当前浮点环境保存在提供的 fenv_t 变量中，设置所有异常不被捕获并清除所有异常粘性标记。可随后执行可能导致出现不需要的异常的代码，并确保清除这些异常的粘性标记。然后，可以调用 feupdateenv()。这将恢复所有异常捕获，并在必要时对其进行调用。

例如，假定有一个 frob() 函数可能导致出现“下溢”或“无效运算”异常（假定捕获这两个异常）。您对“下溢”不感兴趣，但想知道是否进行了无效运算，因此，可以执行以下命令：

```
fenv_t env;
feholdexcept(&env);
frob();
feclearexcept(FE_UNDERFLOW);
feupdateenv(&env);
```

随后，如果 frob() 函数产生“下溢”，feclearexcept() 则会再次将其清除，因此在调用 feupdateenv() 时不会发生捕获。但是，如果 frob() 产生“无效运算”，则在调用 feupdateenv() 时将设置粘性标记，因而会调用捕获处理程序。

此机制是由 C99 提供的，因为 C99 没有指定为各个异常更改异常捕获的方式。更好的方法是使用 __ieee_status() 禁用“下溢”捕获，而将“无效运算”捕获保持启用。这样做的好处是，可以为“无效运算”捕获处理程序提供与无效运算有关的所有信息 （即，对哪些数据执行了哪些运算），并且可以生成该运算的结果。使用 C99 方法时，将在出现异常后调用“无效运算”捕获处理程序，不会接收与异常原因有关的任何信息，并且由于调用时间太晚而无法提供替代结果。

4.2.5 ARM 编译器的 C99 接口扩展

ARM 编译器为 C99 接口提供了一些扩展，以使其能够执行 ARM 浮点环境允许的所有操作。这包括捕获和不捕获个别异常类型，并且还包括安装自定义捕获处理程序。

C99 没有将 fenv_t 和 fexcept_t 类型定义为特殊类型。ARM 编译器将它们定义为相同的结构类型：

```
typedef struct{
    unsigned statusword;
    __ieee_handler_t __invalid_handler;
    __ieee_handler_t __divbyzero_handler;
    __ieee_handler_t __overflow_handler;
    __ieee_handler_t __underflow_handler;
    __ieee_handler_t __inexact_handler;
} fenv_t, fexcept_t;
```

此结构的成员是：

- statusword 是 __ieee_status 函数查看的相同状态变量，并以相同格式进行排列 （请参阅第4-10 页的*__ieee_status()*）。
- 五个函数指针给出了每个异常的捕获处理程序地址。缺省值均为 NULL。这意味着，如果捕获异常，则会执行缺省异常捕获操作。缺省操作是导致生成 **SIGFPE** 信号。

编写自定义异常捕获处理程序

如果要安装自定义异常捕获处理程序，请将其声明为函数，如下所示：

```
__softfp__ieee_value_t myhandler(__ieee_value_t op1,
                                 __ieee_value_t op2,
                                 __ieee_edata_t edata);
```

此函数的参数是：

- op1 和 op2 用于为导致出现异常的运算指定操作数或中间结果：
  - 对于“无效运算”和“除以零”异常，将提供原始操作数。
  - 对于“不精确结果”异常，仅提供以任何方式返回的普通结果。这是在 op1 中提供的。
  - 对于“溢出”异常，将提供中间结果。此结果的计算方法是，计算在指数范围足够大时的运算返回值，然后调整指数以使其符合格式的要求。在单精度型中，将指数调整 192 (0xC0)；在双精度型中，将指数调整 1536 (0x600)。

    如果在将 **double** 转换为 **float** 时发生“溢出”，则以 **double** 格式提供结果、将其舍入为单精度型并且指数偏移 192。
  - 对于“下溢”异常，将生成类似的中间结果，但将偏移值加到指数上，而不是从指数中减去。edata 参数也包含一个标记，用于说明需要向上舍入、向下舍入还是根本不舍入中间结果。

  __ieee_value_t 类型被定义为可传递操作数的所有可能类型的联合，如下所示：

```
typedef union{
    float __f;
    float __s;
    double __d;
    int __i;
    unsigned int __ui;
#if !defined(__STRICT_ANSI__) || (defined(__STDC_VERSION__)
                              && 199901L <= __STDC_VERSION__)
    long long __l;
    unsigned long long __ul;
#endif                                  /* __STRICT_ANSI__ */
    struct { int __word1, __word2; } __str;
} __ieee_value_t;                       /* in/out values passed to traps */
```

注意

如果未使用 --strict 进行编译，并且代码使用旧 __ieee_value_t 定义，则旧代码仍然有效。有关详细信息，请参阅 fenv.h 文件。

- edata 包含一些标记，它们提供了有关所发生的异常以及执行的运算的详细信息。（__ieee_edata_t 类型是 **unsigned int** 的同义词。）
- 函数的返回值用作导致异常的运算的结果。

包含在 edata 中的标记是：

- edata & FE_EX_RDIR 在“下溢”的中间结果向下舍入时不为零，在向上舍入或未舍入时为零。（“不精确结果”位中给出了后两种情况之间的差值。）对于任何其他异常类型，此位没有意义。
- 如果发生了给定*异常*（INVALID、DIVBYZERO、OVERFLOW、UNDERFLOW 或 INEXACT），则 edata & FE_EX_*exception* 不为零。这样，即可：
  - 将相同的处理函数用于多种异常类型（函数可测试这些位以指出应该处理哪种异常）
  - 确定“溢出”和“下溢”中间结果是已舍入的结果还是精确结果。

  由于可以将 FE_EX_INEXACT 位与 FE_EX_OVERFLOW 或 FE_EX_UNDERFLOW 一起进行设置，因此，必须在测试“不精确”结果之前测试“溢出”和“下溢”以确定实际发生的异常类型。
- 如果在执行运算时设置了 FZ 位，则 edata & FE_EX_FLUSHZERO 不为零（请参阅第4-10 页的*__ieee_status()*）。
- edata & FE_EX_ROUND_MASK 给出了应用于运算的舍入模式。这通常与当前舍入模式相同，除非导致异常的运算是始终向零舍入的例程，如 _ffix。可用的舍入模式值是 FE_EX_ROUND_NEAREST、FE_EX_ROUND_PLUSINF、FE_EX_ROUND_MINUSINF 和 FE_EX_ROUND_ZERO。
- edata & FE_EX_INTYPE_*MASK* 将函数操作数的类型指定为表 4-9 中显示的类型值之一。

表 4-9 FE_EX_INTYPE_*MASK* 操作数类型标记

标记	操作数类型
FE_EX_INTYPE_FLOAT	**float**
FE_EX_INTYPE_DOUBLE	**double**
FE_EX_INTYPE_FD	**float double**
FE_EX_INTYPE_DF	**double float**
FE_EX_INTYPE_INT	**int**

表 4-9 FE_EX_INTYPE_*MASK* 操作数类型标记 （续）

标记	操作数类型
FE_EX_INTYPE_UINT	**unsigned int**
FE_EX_INTYPE_LONGLONG	**long long**
FE_EX_INTYPE_ULONGLONG	**无符号 long long**

- edata & FE_EX_OUTTYPE_*MASK* 将函数操作数的类型指定为表 4-10 中显示的类型值之一。

表 4-10 FE_EX_OUTTYPE_*MASK* 操作数类型标记

标记	操作数类型
FE_EX_OUTTYPE_FLOAT	**float**
FE_EX_OUTTYPE_DOUBLE	**double**
FE_EX_OUTTYPE_INT	**int**
FE_EX_OUTTYPE_UINT	**unsigned int**
FE_EX_OUTTYPE_LONGLONG	**long long**
FE_EX_OUTTYPE_ULONGLONG	**无符号 long long**

- edata & FE_EX_FN_*MASK* 将导致异常的运算的特性指定为表 4-11 中显示的运算代码之一。

表 4-11 FE_EX_FN_*MASK* 运算类型标记

标记	运算类型
FE_EX_FN_ADD	相加。
FE_EX_FN_SUB	相减。
FE_EX_FN_MUL	相乘。
FE_EX_FN_DIV	相除。
FE_EX_FN_REM	求余数。
FE_EX_FN_RND	取整。

表 4-11 FE_EX_FN_*MASK* 运算类型标记 （续）

标记	运算类型
FE_EX_FN_SQRT	求平方根。
FE_EX_FN_CMP	比较。
FE_EX_FN_CVT	格式转换。
FE_EX_FN_LOGB	指数获取。
FE_EX_FN_SCALBN	定标。 **注意** FE_EX_INTYPE_MASK 标记仅指定第一个操作数的类型。第二个操作数始终是 **int**。
FE_EX_FN_NEXTAFTER	下一个可表示的数。 **注意** 两个操作数的类型相同。nexttoward 调用可导致将第二个操作数的值更改为与第一个操作数相同类型的值。这不会影响结果。
FE_EX_FN_RAISE	feraiseexcept() 或 feupdateenv() 显式地引发异常。在这种情况下， edata 字中的所有信息几乎都是无效的。

如果是比较运算，则必须以 **int** 方式返回结果，而且必须为表 4-12 中显示的四个值之一。

对于 “比较”和 “转换”之外的所有运算，输入和输出类型是相同的。

表 4-12 FE_EX_CMPRET_*MASK* 比较类型标记

标记	比较
FE_EX_CMPRET_LESS	op1 小于 op2
FE_EX_CMPRET_EQUAL	op1 等于 op2
FE_EX_CMPRET_GREATER	op1 大于 op2
FE_EX_CMPRET_UNORDERED	无法比较 op1 和 op2

示例异常处理程序

示例 4-1 说明了一个自定义异常处理程序。假定将某些 Fortran 代码转换为 C 代码。Fortran 数值标准要求 0 除以 0 得 1，而 IEEE 754 规定 0 除以 0 是“无效运算”，因此，缺省返回无提示 NaN。Fortran 代码可能会依赖于这种行为，让 0 除以 0 返回 1 可能更方便一些，而不是对代码进行修改。

进行编译时，必须选择一种支持异常的浮点模型，例如 `--fpmode=ieee_full` 或 `--fpmode=ieee_fixed`。

安装处理程序之后， 0.0 除以 0.0 返回 1.0。

示例 4-1 自定义异常处理程序

```
#include <fenv.h>
#include <signal.h>
#include <stdio.h>
__softfp __ieee_value_t myhandler(__ieee_value_t op1, __ieee_value_t op2,
                                  __ieee_edata_t edata)
{
    __ieee_value_t ret;
    if ((edata & FE_EX_FN_MASK) == FE_EX_FN_DIV)
    {
        if ((edata & FE_EX_INTYPE_MASK) == FE_EX_INTYPE_FLOAT)
        {
            if (op1.f == 0.0 && op2.f == 0.0)
            {
                ret.f = 1.0;
                return ret;
            }
        }
        if ((edata & FE_EX_INTYPE_MASK) == FE_EX_INTYPE_DOUBLE)
        {
            if (op1.d == 0.0 && op2.d == 0.0)
            {
                ret.d = 1.0;
                return ret;
            }
        }
    }
    /* For all other invalid operations, raise SIGFPE as usual */
    raise(SIGFPE);
}
int main(void)
{
    float i, j, k;
    fenv_t env;
```

```
    fegetenv(&env);
    env.statusword |= FE_IEEE_MASK_INVALID;
    env.invalid_handler = myhandler;
    fesetenv(&env);
    i = 0.0;
    j = 0.0;
    k = i/j;
    printf( “k is %f\n “, k);
}
```

通过信号处理异常捕获

如果捕获了某种异常，但将捕获处理程序地址设置为 NULL，则使用缺省捕获处理程序。

缺省捕获处理程序发出 **SIGFPE** 信号。**SIGFPE** 的缺省处理程序输出一条错误消息并终止程序。

如果捕获 **SIGFPE**，则可以声明信号处理函数，以使用第二个参数指示所发生的浮点异常类型。此功能是为了与 Microsoft 产品保持兼容而提供的。值为 _FPE_INVALID、 _FPE_ZERODIVIDE、 _FPE_OVERFLOW、 _FPE_UNDERFLOW 和 _FPE_INEXACT。这些值是在 float.h 中定义的。例如：

```
void sigfpe(int sig, int etype){
    printf("SIGFPE (%s)\n",
           etype == _FPE_INVALID ? "Invalid Operation" :
           etype == _FPE_ZERODIVIDE ? "Divide by Zero" :
           etype == _FPE_OVERFLOW ? "Overflow" :
           etype == _FPE_UNDERFLOW ? "Underflow" :
           etype == _FPE_INEXACT ? "Inexact Result" :
           "Unknown");
}
signal(SIGFPE, (void(*)(int))sigfpe);
```

要使用此额外信息生成您自己的 **SIGFPE** 信号，可以调用 __rt_raise() 函数以替代 ISO 函数 raise()。在第4-24 页的示例 4-1 中，替代的是：

```
    raise(SIGFPE);
```

最好使用以下方式进行编码：

```
    __rt_raise(SIGFPE, _FPE_INVALID);
```

在 rt_misc.h 中声明 __rt_raise()。

4.3 数学库 mathlib

除了 ISO C 标准定义的函数以外，mathlib 还提供了以下 C99 函数和宏。它们是对浮点库 fplib 提供的函数和宏的补充 （请参阅第4-2 页的*软件浮点库 fplib*）：

- 第4-27 页的*确定浮点数类型 (fpclassify)*
- 第4-27 页的*确定一个数是否为有限数 (isfinite)*
- 第4-27 页的*确定一个数是否为无穷大 (isinf)*
- 第4-28 页的*确定一个数是否为 NaN (isnan)*
- 第4-28 页的*确定一个数是否为正规数 (isnormal)*
- 第4-28 页的*返回一个数的符号位 (signbit)*
- 第4-29 页的*复制符号函数 （copysign、 copysignf）*
- 第4-29 页的*比较宏 （isgreater、 isgreaterequal、 isless、 islessequal、 islessgreater、 isunordered）*
- 第4-30 页的*反双曲函数 （acosh、 asinh、 atanh）*
- 第4-30 页的*立方根 (cbrt)*
- 第4-30 页的*误差函数 （erf、 erfc）*
- 第4-31 页的*exp(x) 减 1 (expm1)*
- 第4-31 页的*求斜边函数 (hypot)*
- *gamma 函数的对数* 第4-32 页的*lgamma*
- 第4-32 页的*x 加 1 的对数 (log1p)*
- 第4-33 页的*IEEE 754 求余数函数 (remainder)*
- 第4-34 页的*IEEE 取整运算 (rint)*

Mathlib 还提供了下列非标准函数：

- 第4-31 页的*gamma 函数 （gamma、 gamma_r）*
- 第4-31 页的*第一类 Bessel 函数 （j0、 j1、 jn）*
- *gamma 函数的对数* 第4-32 页的*lgamma_r*
- 第4-34 页的*返回一个数的小数部分 (significand)*
- 第4-34 页的*第二类 Bessel 函数 （y0、 y1、 yn）*

4.3.1 mathlib 中的范围缩小函数

mathlib 中的三角函数可通过使用范围缩小函数，将较大的参数放在 0 到 2π 的范围内。ARM 编译器提供了两个不同的范围缩小函数。其中一个函数精确到*任意*输入值的最后一位，但比另一个函数大，并且速度也更慢一些。对于绝大多数应用场合来说，使用第二个函数就足够可靠了，而且该函数更快更小。

缺省情况下，使用这种又快又小的范围缩小函数。要选择那个更精确的函数，请使用以下任一方法：

- C 中的 `#pragma import (__use_accurate_range_reduction)`
- 汇编语言中的 `IMPORT __use_accurate_range_reduction`。

4.3.2 确定浮点数类型 (fpclassify)

此宏根据类型对其自变量值进行分类。

```
int fpclassify(real-floating x);
```

其中， *real-floating* 可以为 **float**、**double** 或 **long double** 类型。

此处，`fpclassify` 返回以下值之一：

- `FP_INFINITE` （如果 `x` 为无穷大）
- `FP_NAN` （如果 `x` 为 NaN）
- `FP_NORMAL` （如果 `x` 为正规数）
- `FP_SUBNORMAL` （如果 `x` 为次正规数）
- `FP_ZERO` （如果 `x` 为零）。

它不会导致出现任何错误或异常。

4.3.3 确定一个数是否为有限数 (isfinite)

此宏确定其参数是否具有一个有限的值。

```
int isfinite(real-floating x);
```

其中， *real-floating* 可以为 **float**、**double** 或 **long double** 类型。

此处，`isfinite` 返回以下值：

- 非零值 （如果 `x` 为有限数）
- `0` （如果 `x` 为无穷大或 NaN）。

它不会导致出现任何错误或异常。

4.3.4 确定一个数是否为无穷大 (isinf)

此宏确定其参数是否为无穷大。

```
int isinf(real-floating x);
```

其中， *real-floating* 可以为 **float**、**double** 或 **long double** 类型。

此处，isinf 返回以下值：
- 非零值 （如果 x 为无穷大）
- 0 （如果 x 为有限数或 NaN）。

它不会导致出现任何错误或异常。

4.3.5 确定一个数是否为 NaN (isnan)

此宏确定其自变量值是否为 NaN。

```
int isnan(real-floating x);
```

其中， *real-floating* 可以为 **float**、**double** 或 **long double** 类型。

此处，isnan 返回以下值：
- 非零值 （如果 x 为 NaN）
- 否则为 0。

它不会导致出现任何错误或异常。

4.3.6 确定一个数是否为正规数 (isnormal)

此宏确定其自变量值是否为正规数。

```
int isnormal(real-floating x);
```

其中， *real-floating* 可以为 **float**、**double** 或 **long double** 类型。

此处，isnormal 返回以下值：
- nonzero （如果 x 为正规数）
- 0 （如果 x 为无穷大、 NaN、次正规数或零）。

它不会导致出现任何错误或异常。

4.3.7 返回一个数的符号位 (signbit)

此宏确定其自变量值的符号是否为负号。

```
int signbit(real-floating x);
```

其中， *real-floating* 可以为 **float**、**double** 或 **long double** 类型。

此处，signbit 返回以下值：
- 1 （如果 x 为负数）

- 0（如果 x 为正数）。

signbit 返回所有 x 值（包括零和 NaN）的符号值。

它不会导致出现任何错误或异常。

4.3.8 复制符号函数（copysign、copysignf）

这些函数将 x 的符号位替换为 y 的符号位。

```
double copysign(double x, double y);
float copysignf(float x, float y);
```

这些函数将所有浮点数视为有符号数（包括零和 NaN），并且不会导致出现任何错误或异常。

4.3.9 比较宏（isgreater、isgreaterequal、isless、islessequal、islessgreater、isunordered）

这些宏比较 x 和 y，如表 4-13 中所示。

```
int isgreater(real-floating x, real-floating y);
int isgreaterequal(real-floating x, real-floating y);
int isless(real-floating x, real-floating y);
int islessequal(real-floating x, real-floating y);
int islessgreater(real-floating x, real-floating y);
int isunordered(real-floating x, real-floating y);
```

其中，*real-floating* 可以为 **float**、**double** 或 **long double** 类型。

与在所有 NaN 上产生异常的等效关系运算符不同，这些宏不会导致出现任何错误或异常。

表 4-13 比较宏

宏	比较
isgreater	x 大于 y
isgreaterequal	x 大于或等于 y
isless	x 小于 y

表 4-13 比较宏 （续）

宏	比较
islessequal	x 小于或等于 y
islessgreater	x 小于或大于 y
isunordered	相对于 y， x 是无序结果

4.3.10 反双曲函数 （acosh、 asinh、 atanh）

这些函数是 cosh、 sinh 和 tanh 的反函数。

```
double acosh(double x);
double asinh(double x);
double atanh(double x);
```

其中：

- acosh 始终可以从两个返回值 （一个为正值，一个为负值）中进行选择，这是因为 cosh 是对称函数（即应用于 *x* 或 *-x* 时返回同一值）。它选择正的结果值。
- 如果使用小于 1.0 的自变量进行调用， acosh 将返回 EDOM 错误。
- 如果使用绝对值超过 1.0 的自变量进行调用， atanh 将返回 EDOM 错误。

4.3.11 立方根 (cbrt)

此函数返回其自变量的立方根。

```
double cbrt(double x);
```

4.3.12 误差函数 （erf、 erfc）

这些函数计算与正态分布有关的标准统计误差函数：

```
double erf(double x);
double erfc(double x);
```

其中：

- erf 计算 *x* 的普通误差函数

- erfc 计算 1 减 erf(*x*)的结果。如果 *x* 很大，则最好使用 erfc(*x*) 而不是 1-erf(*x*)，因为这样可以得到更准确的结果。

4.3.13 exp(*x*) 减 1 (expm1)

此函数计算 e^x 减去 1 的结果。如果 *x* 非常接近于零，则最好使用 expm1(x) 而不是 exp(x)-1，因为 expm1 可返回更准确的值。

```
double expm1(double x);
```

4.3.14 gamma 函数（gamma、 gamma_r）

这两个函数都计算 gamma 函数的对数。它们是 lgamma 和 lgamma_r 的同义词（请参阅第4-32 页的*gamma 函数的对数*）。

```
double gamma(double x);
double gamma_r(double x, int *);
```

注意

尽管函数的名称不同，但这些函数都计算 gamma 函数的对数，而不是 gamma 函数本身。

注意

已不提倡使用这些函数。

4.3.15 求斜边函数 (hypot)

此函数计算直角三角形的斜边长度，三角形的两条边的长度分别是 *x* 和 *y*。同理，它计算笛卡儿坐标中的向量 (*x,y*) 的长度。最好使用 hypot(x,y) 而不是 sqrt(x*x+y*y)，因为一些 *x* 和 *y* 值可能会导致 x * x + y * y 溢出，即使其平方根不会溢出。

```
double hypot(double x, double y);
```

如果结果不能用 **double** 表示， hypot 将返回 ERANGE 错误。

4.3.16 第一类 Bessel 函数（j0、 j1、 jn）

这些函数计算第一类 Bessel 函数。j0 和 j1 分别计算 0 和 1 阶函数。j*n* 计算 *n* 阶函数。

```
double j0(double x);
double j1(double x);
double jn(int n, double x);
```

如果 *x* 的绝对值超过 π 乘以 2^{52}，则这些函数返回 ERANGE 错误，表示结果已完全失去意义。

注意

已不提倡使用这些函数。

4.3.17 gamma 函数的对数

这些函数计算 *x* 的 gamma 函数的绝对值的对数。将单独返回函数的符号，因此，这两个函数可用于计算 *x* 的实际 gamma 函数。

如果结果太大而不能用 **double** 表示，这两个函数将返回 ERANGE 错误。

如果 *x* 为零或负整数，这两个函数将返回 EDOM 错误。

lgamma

```
double lgamma(double x);
```

lgamma 在全局变量 _signgam 中返回 *x* 的 gamma 函数的符号。

lgamma_r

```
double lgamma_r(double x, int *sign);
```

lgamma_r 在用户变量中返回该符号，用户变量的地址是在 sign 参数中传递的。在这两种情况下，该值是 +1 或 – 1。

注意

已不提倡使用 lgamma_r()。

4.3.18 x 加 1 的对数 (log1p)

此函数计算 *x* + 1 的自然对数。与 expm1 一样，最好使用此函数而不是 log(x+1)，因为在 *x* 接近于零时此函数更准确。

```
double log1p(double x);
```

4.3.19 IEEE 754 求余数函数 (remainder)

此函数执行 IEEE 754 求余数运算。它是 _drem 的同义词 （请参阅第4-3 页的*特定格式数字的算法*）。

```
double remainder(double x, double y);
```

4.3.20 IEEE 取整运算 (rint)

此函数执行 IEEE 754 取整运算。它是 _drnd 的同义词 （请参阅第4-3 页的*特定格式数字的算法*）。

```
double rint(double x);
```

4.3.21 返回一个数的小数部分 (significand)

此函数将 *x* 的小数部分作为 1.0 和 2.0 之间的数 （不包括 2.0）返回。

```
double significand(double x);
```

注意

已不提倡使用此函数。

4.3.22 第二类 Bessel 函数 （y0、 y1、 yn）

这些函数计算第二类 Bessel 函数。y0 和 y1 分别计算 0 和 1 阶函数。yn 计算 *n* 阶函数。

```
double y0(double x);
double y1(double x);
double yn(int, double);
```

如果 *x* 为正数并超过 π 乘以 2^{52}，则这些函数返回 ERANGE 错误，表示结果已完全失去意义。

注意

已不提倡使用这些函数。

4.4 IEEE 754 算法

ARM 浮点环境实现了 IEEE 754 二进制浮点算法标准。本节简要说明了 ARM 编译器实现的这一标准。

本节包括以下内容：

- *基本数据类型*
- 第4-39 页的*算法和舍入*
- 第4-39 页的*异常*

4.4.1 基本数据类型

ARM 浮点值存储在以下两种数据类型之一中：*单精度* 和 *双精度*。在本文档中，这些类型称为 **float** 和 **double**。这些是相应的 C 类型。

单精度

float 值为 32 位宽。其结构显示在图 4-3 中。

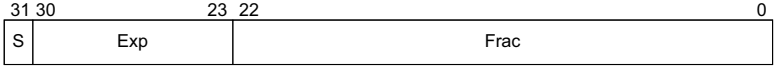

图 4-3 IEEE 754 单精度浮点格式

S 字段给出一个数的符号。0 表示正数， 1 表示负数。

Exp 字段给出一个数的指数，该指数是 2 的幂。其*偏移为* 0x7F (127)，以使很小的数的指数接近于零，很大的数的指数接近于 0xFF (255)。

因此，举例来说：

- 如果 *Exp* = 0x7D (125)，则数在 0.25 和 0.5 之间 （不包括 0.5）
- 如果 *Exp* = 0x7E (126)，则数在 0.5 和 1.0 之间 （不包括 1.0）
- 如果 *Exp* = 0x7F (127)，则数在 1.0 和 2.0 之间 （不包括 2.0）
- 如果 *Exp* = 0x80 (128)，则数在 2.0 和 4.0 之间 （不包括 4.0）
- 如果 *Exp* = 0x81 (129)，则数在 4.0 和 8.0 之间 （不包括 8.0）。

Frac 字段给出数的小数部分。通常，它前面带有一个隐式 1 位，不会存储此位以节省空间。

因此，举例来说，如果 *Exp* 是 0x7F：

- 如果 *Frac* = 00000000000000000000000 （二进制），数为 1.0
- 如果 *Frac* = 10000000000000000000000 （二进制），数为 1.5

- 如果 *Frac* = 01000000000000000000000 （二进制），数为 1.25
- 如果 *Frac* = 11000000000000000000000 （二进制），数为 1.75。

因此，在通常情况下，采用这种格式的位模式数值由下面的公式给出：

$(-1)^{S} * 2^{(Exp - 0x7F)} * (1 + Frac * 2^{-23})$

使用这种格式存储的数称为*规格化*数。

最大指数值 255 和最小指数值 0 是特例。指数 255 用于表示无穷大并存储 NaN 值。如果用一个数除以零，或者计算的值太大而不能以这种格式存储，都可能得到无穷大。NaN 值用于特殊用途。可通过将 Exp 设置为 255 而将 Frac 设置为全零来存储无穷大。如果 Exp 是 255 而 Frac 不为零，则位模式表示 NaN。

指数 0 用于以特殊方式表示很小的数。如果 *Exp* 为零，则 Frac 字段前面没有隐式 1。这意味着，该格式可通过将 Exp 和 Frac 均设置为全零位来存储 0.0。这还意味着，太小而不能用 Exp >= 1 存储的数用低于普通 23 位的精度进行存储。这些数称为*非正规数*。

双精度

double 值为 64 位宽。图 4-4 显示了其结构。

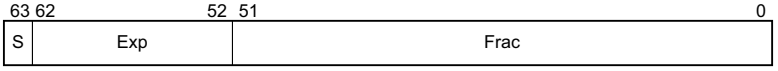

图 4-4 IEEE 754 双精度浮点格式

与前面一样，S 是符号，Exp 是指数，而 Frac 是小数。在大多数讨论中，**float** 值保持为真，除非：

- Exp 字段偏移为 0x3FF (1023)，而不是 0x7F，因此，1.0 和 2.0 之间的数的 Exp 字段偏移为 0x3FF。
- 用于表示无穷大和 NaN 的 Exp 值是 0x7FF (2047) 而不是 0xFF。

示例值

本节提供了示例浮点值：

- *单精度浮点值*
- 第4-38 页的*双精度浮点值*

单精度浮点值

表 4-14 中给出了一些示例 **float** 位模式及其数学值。

表 4-14 示例单精度浮点值

浮点值	S	Exp	Frac	数学值	说明[a]
0x3F800000	0	0x7F	000...000	1.0	-
0xBF800000	1	0x7F	000...000	-1.0	-
0x3F800001	0	0x7F	000...001	1.000 000 119	a
0x3F400000	0	0x7E	100...000	0.75	-
0x00800000	0	0x01	000...000	1.18×10^{-38}	b
0x00000001	0	0x00	000...001	1.40×10^{-45}	c
0x7F7FFFFF	0	0xFE	111...111	3.40×10^{38}	d
0x7F800000	0	0xFF	000...000	正无穷大	-
0xFF800000	1	0xFF	000...000	负无穷大	-
0x00000000	0	0x00	000...000	0.0	e
0x7F800001	0	0xFF	000...001	信号 NaN	f
0x7FC00000	0	0xFF	100...000	无提示 NaN	f

a. 有关详细信息，请参阅第4-38 页的*示例浮点值注释*。

双精度浮点值

表 4-15 中给出了一些示例 **double** 位模式及其数学值。

表 4-15 示例双精度浮点值

双精度值	S	Exp	Frac	数学值	说明[a]
0x3FF00000 00000000	0	0x3FF	000...000	1.0	-
0xBFF00000 00000000	1	0x3FF	000...000	-1.0	-
0x3FF00000 00000001	0	0x3FF	000...001	1.000 000 000 000 000 222	a
0x3FE80000 00000000	0	0x3FE	100...000	0.75	-
0x00100000 00000000	0	0x001	000...000	$2.23*10^{-308}$	b
0x00000000 00000001	0	0x000	000...001	$4.94*10^{-324}$	c
0x7FEFFFFF FFFFFFFF	0	0x7FE	111...111	$1.80*10^{308}$	d
0x7FF00000 00000000	0	0x7FF	000...000	正无穷大	-
0xFFF00000 00000000	1	0x7FF	000...000	负无穷大	-
0x00000000 00000000	0	0x000	000...000	0.0	e
0x7FF00000 00000001	0	0x7FF	000...001	信号 NaN	f
0x7FF80000 00000000	0	0x7FF	100...000	无提示 NaN	f

a. 有关详细信息，请参阅*示例浮点值注释*。

示例浮点值注释

a 可看作大于 1.0 的最小可表示数。该数与 1.0 之间的差值称为*机器最小数*。**float** 中为 0.000 000 119，**double** 中为 0.000 000 000 000 000 222。可以从机器最小数中大致了解该格式可以记录的有效数字个数。**float** 可以包含 6 或 7 位。**double** 可以包含 15 或 16 位。

b 可以在每种格式中表示为规格化数的最小值。比该值小的数可存储为非正规数，但不能以相当的精度进行存储。

c 可与零相区分的最小正数。这是该格式的绝对下限。

d 可存储的最大有限数。如果尝试通过相加或相乘增大此数，则会导致溢出并生成无穷大（通常情况下）。

e 零。严格地说，它们表示正零。尽管比较运算 （如 == 和 !=）报告这两种类型的零相等，但一些运算以不同方式处理符号位为 1 的零（负零）。

f 共有两种类型的 NaN：信号 NaN 和无提示 NaN。对于无提示 NaN， Frac 的第一位是 1；对于信号 NaN， Frac 的第一位是 0。差别在于，信号 NaN 导致出现异常 （请参阅*异常*），而无提示 NaN 不会导致出现异常。

4.4.2 算法和舍入

算法的通常执行方式为，计算运算结果时将这些结果视为精确存储的值 （无限精度），然后再舍入结果以符合格式的要求。除那些结果本来就已完全符合格式要求的运算 （如 1.0 加 1.0）之外，正确结果通常在可用该格式表示的两个数之间。系统随后选择这两个数中的一个作为舍入结果。系统使用以下方法之一：

舍入到最接近的数

系统选择两个可能输出值中较接近的一个。如果正确结果恰好在两个数中间，系统将选择 Frac 最低有效位为 0 的数。此行为 （舍入到偶数）可防止生成各种不希望出现的结果。

这是应用程序启动时的缺省模式。它是普通浮点库支持的唯一模式。硬件浮点环境和增强型浮点库支持所有四种舍入模式。请参阅第 2-111 页的*库命名约定*。

向上舍入或向正无穷大舍入

系统选择两个可能输出值中较大的一个，即，当它们为正数时，指离零较远的一个；当它们为负数时，指离零较近的一个。

向下舍入或向负无穷大舍入

系统选择两个可能输出值中较小的一个，即，当它们为正数时，指离零较近的一个；当它们为负数时，指离零较远的一个。

向零舍入、分割或截断

在所有情况下，系统选择离零较近的输出值。

4.4.3 异常

浮点算术运算可能会遇到各种问题。例如，计算结果可能太大或太小而不符合格式要求，或者无法计算出结果 （如试图求负数的平方根或用零除以零）。这些称为异常，因为它们表示例外或异常情况。

ARM 浮点环境可使用多种方法处理异常。

忽略异常

系统生成并返回似是而非的运算结果。例如，负数的平方根可能会生成 NaN，而试图计算太大而不符合格式要求的值可能会生成无穷大。如果发生异常并将其忽略，则会在浮点状态字中设置一个标记，指示在过去某一时刻出现错误。

捕获异常

这意味着，在发生异常时，将运行一个称为捕获处理程序的代码块。系统提供了一个缺省捕获处理程序，它输出一条错误消息并终止应用程序。但是，您可以提供自己的捕获处理程序，按照所选择的方式消除异常情况。捕获处理程序甚至可以提供从运算返回的结果。

例如，如果算法可方便地假定 0 除以 0 得 1，则可以为“无效运算”异常提供一个自定义捕获处理程序，以识别这种特殊情况并替换所需的结果。

异常类型

ARM 浮点环境可识别以下类型的异常：

- 如果运算没有合理的结果，则会发生“无效运算”异常。发生这种异常的原因可能是：
  - 对信号 NaN 执行任何运算，最简单的运算 （复制和更改符号）除外
  - 将正无穷大和负无穷大相加，或将无穷大与其自身相减
  - 将无穷大乘以零
  - 将零除以零，或将无穷大除以无穷大
  - 求任意数除以零或无穷大除以任意数的余数
  - 求负数的平方根 （不包括负零）
  - 在结果不符合格式要求时，将浮点数转换为整数
  - 在两个数中的一个是 NaN 时进行比较。

  如果未捕获“无效运算”异常，所有这些运算将返回无提示 NaN；到整数的转换除外，它返回的是零 （因为整数中没有无提示 NaN）。
- 如果将非零的有限数除以零，则会发生“除以零”异常。（将零除以零发生“无效运算”异常。将无穷大除以零有效并返回无穷大。）

  如果未捕获 “除以零”，该运算将返回无穷大。

- 如果运算结果太大而不符合格式要求，则会发生 “溢出”异常。例如，如果将最大可表示数 （第4-37 页的表 4-14 中标记为 d）与自身相加，则会发生这种异常。

  如果未捕获“溢出”，运算将返回无穷大或最大有限数，具体取决于舍入模式。

- 如果运算结果太小而无法表示为规格化数（Exp 至少为 1），则会发生“下溢”异常。

  导致发生 “下溢”的情况取决于是否将其捕获：

  — 如果捕获“下溢”，只要结果太小而不能表示为规格化数，就会发生“下溢”异常。

  — 如果未捕获“下溢”，仅当需要舍入结果时才会发生“下溢”异常。因此，举例来说，将 **float** 数 0x00800000 除以 2 时不会生成“下溢”信号，因为结果 (0x00400000) 是精确的。但是，如果尝试将 float 数 0x00000001 乘以 1.5，则会生成 “下溢”信号。

  （对于熟悉 IEEE 754 规范的读者， ARM 编译器中选择的实现选项可用于在舍入之前检测极小值，并检测不精确结果的精度下降情况。）

  如果未捕获 “下溢”，则根据当前舍入模式，将结果舍入到两个最接近的可表示非正规数中的一个。将忽略精度下降，系统返回可能的最佳结果。

  — 只要运算结果需要舍入，就会发生“不精确结果”异常。如果需要在软件的每个运算中都检测此异常，则会使速度显著下降，因此，普通浮点库不支持“不精确结果”异常。增强型浮点库和硬件浮点系统均支持 “不精确结果”。

  如果未捕获 “不精确结果”，系统将按通常方式舍入结果。

  如果未捕获“溢出”和“下溢”中的任何一个，它们也会设置“不精确结果”标记。

缺省情况下，不捕获所有异常。