RVM微控制器用户态库
技术参考手册

版本 R3T1

突越·中生·菊石（七阶）

Mutatus·Mesozoa·Ammonite(R.VII)

M7M2(Ammonite) R3T1

微控制器用户态库（三版一型）

技术手册

系统特性

高性能实时响应

·原生进程提供优秀的实时性能

·虚拟机进程提供优秀的兼容性和执行效率

全面的体系架构支持

·可简易地在多种架构之间移植

·在微控制器上支持准虚拟化

高安全性

·虚拟机之间硬件隔离

·支持故障重启功能

全面的生态环境

·支持虚拟化绝大多数操作系统

·原系统应用程序无需修改即可使用

高易用性

·提供图形化虚拟机配置界面（G7S2/GSC）

目录

表目录

图目录

版本历史

版本	日期（年-月-日）	说明
R1T1	2018-03-09	初始发布
R2T1	2018-09-18	增加了移植说明

第 1 章 概述

1.1 简介

在现代微控制器（MCU）应用中，人们对灵活性、安全性和可靠性的要求日益提高。越来越多的微控制器应用要求多个互不信任的来源的应用程序一起运行，也可能同时要求实时可靠部分[1]、性能优先部分[2]和某些对信息安全有要求的应用[3]一起运行，而且互相之间不能干扰。在某些应用程序宕机时，要求它们能够单独重启，并且在此过程中还要求保证那些直接控制物理机电输出的应用程序不受干扰。

由于传统实时操作系统（Real-Time Operating System，RTOS）的内核和应用程序通常静态链接在一起，而且互相之间没有内存保护隔离，因此达不到应用间信息隔离的要求；一旦其中一个应用程序崩溃而破坏内核数据，其他所有应用程序必然同时崩溃。一部分 RTOS 号称提供了内存保护，但是其具体实现机理表明其内存保护仅仅能够提供一定程度的应用间可靠性而无法提供应用间安全性，不适合现代微控制器应用的信息安全要求。此外，传统 RTOS 的内核服务[4]是在高优先级应用和低优先级应用之间共用的，一旦低优先级应用调用大量的内核服务，势必影响高优先级应用内核服务的响应，从而造成时间干扰问题[1]。

RVM（Real-time Virtual machine Monitor）就是针对上述问题而提出的、面向高性能 MCU 的、基于 RME 微内核的用户态库。它是运行在 RME 操作系统上的一个宿主型虚拟机监视器，使得在微控制器平台上运行多个虚拟机成为可能，也同时满足用户运行原生 RME 应用程序的需求。这样，实时和非实时应用一起运行的需求，在实时应用和非实时应用之间不会产生时间干扰。它也能够使多来源应用程序在同一芯片上运行而不发生信息安全问题。

由于 RVM 强制隔离了系统中安全的部分和不安全的部分，这两部分就可以用不同的标准分开开发和认证。这方便了调试，也节省了开发和认证成本，尤其是当使用到那些较为复杂、认证程度较低和实时性较差的的软件包如用户图形界面（Graphical User Interface，GUI）和高级语言虚拟机（Python、Java 虚拟机）等的时候。此外，在老平台上已得到认证的微控制器应用程序可以作为一个虚拟机直接运行在 RVM 上，无需重新认证；这进一步节省了成本。

本手册从用户的角度提供了 RVM 的功能描述。关于各个架构的具体使用，请参看各个架构相应的手册。在本手册中，我们先简要回顾关于虚拟化的若干概念，然后分章节介绍 RVM 的特性和 API。

1.1.1 微控制器的特点

在实际应用中，MCU 系统占据了实时系统中的相当部分。它常常有如下几个特点：

[1] 如电机控制等

[2] 如网络协议栈、高级语言虚拟机等

[3] 如加解密算法等

[4] 如内核定时器等

1.1.1.1 深度嵌入

微控制器的应用场合往往和设备本身融为一体，用户通常无法直接感知到此类系统的存在。微控制器的应用程序[1]一旦写入，往往在整个生命周期中不再更改，或者很少更改。即便有升级的需求，往往也由厂家进行升级。

1.1.1.2 高度可靠

微控制器应用程序对于可靠性有近乎偏执的追求，通常而言任何稍大的系统故障都会引起相对严重的后果。从小型电压力锅到大型起重机，这些系统都不允许发生严重错误，否则会造成重大财产损失或人身伤亡。此外，对于应用的实时性也有非常高的要求，它们通常要么要求整个系统都是硬实时系统，或者系统中直接控制物理系统的那部分为硬实时系统。

1.1.1.3 资源有限

由于微控制器本身资源不多，因此高效地利用它们就非常重要了。微控制器所使用的软件无论是在编写上和编译上总体都为空间复杂度优化，只有小部分对性能和功能至关重要的程序使用时间复杂度优化。

1.1.1.4 静态分配和链接

与微处理器不同，由于深度嵌入的原因，微控制器应用中的绝大部分资源，包括内存和设备在内，都是在系统上电时创建并分配的。通常而言，在微控制器内使用过多的动态特性是不明智的，因为动态特性不仅会对系统的实时响应造成压力，另一方面也会增加功能成功执行的不确定性。如果某些至关重要的功能和某个普通功能都依赖于动态内存分配，那么当普通功能耗尽内存时，可能导致重要功能的内存分配失败。基于同样的或类似的理由，在微控制器中绝大多数代码都是静态链接的，一般不使用动态链接。

1.1.1.5 没有虚拟地址空间

与微处理器不同，微控制器通常都不具备内存管理单元（Memory Management Unit，MMU），因此无法实现物理地址到虚拟地址的转换。但是，RVM 也支持使用内存保护单元（Memory Protection Unit，MPU），并且大量的中高端微控制器都具备 MPU。与 MMU 不同，MPU 往往仅支持保护一段或几段内存范围，有些还有很复杂的地址对齐限制，因此内存管理功能在微控制器上往往是有很大局限性的。

1.1.1.6 单核处理器

如今，绝大多数的微处理器都是多核设计。然而，与微处理器不同，微控制器通常均为单核设计，因此无需考虑很多竞争冒险问题。这可以进一步简化微控制器用户态库的设计。考虑到的确有少部分微

[1] 通常称为“固件”

控制器采取多核设计或者甚至不对称多核设计，RVM 也支持多核运行环境。当然，这首先需要底层的 RME 被正确配置为支持多核。

考虑到以上几点，在设计 RVM 时，所有的内核对象都会在系统启动时创建完全，并且在此时，系统中可用内核对象的上限也就决定了。各个虚拟机则被固化于微控制器的片上非易失性存储器中。

1.1.2 软件版权与许可证

综合考虑到微控制器应用、深度嵌入式应用和传统应用对开源系统的不同要求，RVM 微控制器库所采用的许可证为 Unlicensed，但是对一些特殊情况[1]使用特殊的规定。这些特殊规定是就事论事的，对于每一种可能情况的具体条款都会有不同。

1.1.3 主要参考系统

RTOS 通用服务主要参考了如下系统的实现：

RMProkaron（@EDI）

FreeRTOS（@Real-Time Engineering LTD）

Contiki（@The Contiki Community）

虚拟机监视器的设计参考了如下系统的实现：

Nova（@TU Dresden）

XEN（@Cambridge University）

KVM（@The Linux Foundation）

其他各章的参考文献和参考资料在该章列出。

1.2 前言

作为一个虚拟机监视器，RVM 依赖于底层 RME 操作系统提供的内核服务来维持运行。RVM 采用准虚拟化技术，最大限度节省虚拟化开销，典型情况下仅增加约 1%的 CPU 开销。

RVM 完全支持低功耗技术和 Tick-Less 技术，可以实现虚拟化系统的各个分区处于无节拍工作，最大限度减少电力需求。对于那些对确定性有很强要求的应用，该项功能也可以关闭而使用传统的嘀嗒模式。

对于 RME 原生进程应用，其响应时间和 FreeRTOS 一档小型 RTOS 的响应时间相若，完全能够满足那些对实时性要求最苛刻的应用。

对于 RVM 宿主型虚拟机[2]，中断响应时间和线程切换时间将会增加到原小型 RTOS 的约 4 倍。值得注意的是，RVM 本身并不提供任何 POSIX 支持；POSIX 支持是依赖于所移植的操作系统本身的。幸运的是，很多 RTOS 都具备 POSIX 支持，因此直接虚拟化它们就可以使用其 POSIX 功能了。

[1] 比如安防器材、军工系统、航空航天装备、电力系统和医疗器材等

[2] 虚拟机分类见下文所述

RME 和 RVM 在全功能配置下共占用最小 16kB 内存[1]，外加固定占用 48kB 程序存储器，适合中高性能微控制器使用。此外，如果多个 RVM 虚拟机同时使用网络协议栈等组件，使用本技术甚至可节省内存，因为网络协议栈等共享组件可以只在系统中创建一个副本，而且多个操作系统的底层切换代码和管理逻辑由于完全被抽象出去，因此在系统中只有一个副本。

在此我们回顾一下虚拟化的种类。在本手册中，我们把虚拟机按照其运行平台分成三类，分别称为原生型（I 型）、宿主型（II 型）和容器型（III 型）。同时，依照虚拟机支持虚拟化方式的不同，又可以分为全虚拟化型（Full Virtualization，FV）和准虚拟化型（Para-Virtualization，PV）。这两种分类方法是互相独立的。

1.2.1 原生型虚拟机监视器

原生型虚拟机监视器作为底层软件直接运行在硬件上，创造出多个物理机的实例。多个虚拟机在这些物理机的实例上运行。通常而言，这类虚拟机监视器的性能较高，但是自身不具备虚拟机管理功能，需要 0 域[2]对其他虚拟机进行管理。此类虚拟机监视器有 KVM、Hyper-V、ESX Server、Xen、XtratuM 等。

1.2.2 宿主型虚拟机监视器

原生型虚拟机监视器作为应用程序运行在操作系统上，创造出多个物理机的实例。多个虚拟机在这些物理机的实例上运行。通常而言，这类虚拟机监视器的性能较低，但是具备较好的虚拟机管理功能和灵活性。虚拟机均无特权，互相之间无法进行管理。此类虚拟机监视器有 VMware Workstation、Virtual Box、QEMU[3]等。RVM 是在 RME 上开发的一个应用程序，因此也属于宿主型虚拟机监视器。

1.2.3 容器型虚拟机监视器

容器型虚拟机监视器是由操作系统本身提供的功能，创造出多个相互隔离的操作系统资源分配空间的独立实例。多个应用程序直接在这些资源分配空间实例上运行。通常而言，这类虚拟机监视器的性能最高，虚拟化的内存和时间开销最低，但是它要求应用程序必须是针对它提供的接口编写的。通常而言，这接口与提供了容器型虚拟机监视器功能的操作系统自身的系统调用一致。它具备最低的管理和运行成本，但灵活性最差。此类虚拟机监视器有 Docker、Pouch、RKT、LXC 等。当 RVM 导致的性能开销被认为不合适时，由于 RME 操作系统自身也原生具备容器型虚拟机监视器的能力，也可以把 RVM 提供的用户态库用来开发原生的容器型虚拟机应用。

1.2.4 全虚拟化型虚拟机监视器

[1] 此时可支持 4 个虚拟机，若分配 128kB 内存则可支持 32 个虚拟机

[2] 也即第一个启动的、具有管理特权的虚拟机

[3] 不包括其 KVM 版本

全虚拟化型虚拟机监视器是有能力创建和原裸机完全一致的硬件实例的虚拟机监视器。通常而言，这需要 MMU 的介入来进行地址转换，而且对特权指令要进行二进制翻译、陷阱重定向或专用硬件[1]处理，对于 I/O 也需要特殊硬件功能[2]来重定向。此类虚拟机在 MCU 环境上难于实现，因为微控制器通常都不提供 MMU，也不提供相应的虚拟化硬件进行重定向功能。此类虚拟机监视器有 VMware Workstation、Virtual Box、QEMU、KVM、Hyper-V、ESX Server 等。

1.2.5 准虚拟化型虚拟机监视器

全虚拟化型虚拟机监视器是创建了和原裸机有些许差别的的硬件实例的虚拟机监视器。它使用超调用（Hypercall）处理特权指令和特殊 I/O，并借此避开那些复杂、不必要的全虚拟化必须提供的细节。此类虚拟机在 MCU 环境上能够以较高的性能实现。此类虚拟机监视器有 Xen 等。RVM 也是一个准虚拟化型虚拟机监视器，通过超调用处理系统功能。

1.3 RVM 的组成部分

一个 RVM 实例由 RME 原生进程和 RVM 虚拟机两个部分组成。

1.3.1 RME 原生进程

通常而言，RME 原生进程主要用来接受从中断向量来的信号，并对它们加以初步、快速地处理后转送信号出去，主要用于 RME 的用户态驱动程序。这些线程运行在用户态的多个进程之中，并且其时间片都是无限的。这一部分高度安全的线程在 RME 中拥有最高的优先级，并且和虚拟机完全隔离开来。即使虚拟机监视器和所有的虚拟机都已崩溃，这些线程仍能保证正常运行，从而维护被控设备的安全。

由于 RME 原生进程的资源占用最低，因此在那些资源不甚宽裕的处理器上，仅使用 RME 原生进程来编写整个应用程序也是可能的。但是，在 RME 原生进程中只能直接调用 RME 的内核系统调用，其编程所受限制相对较多。如果任何一个 RME 原生进程发生错误，那么代表系统中出现了严重的问题，整个系统将被重启。

1.3.2 RVM 虚拟机

RVM 虚拟机部分负责模拟小型 RTOS 运行时依赖的硬件抽象层，并且藉由此在 RVM 上运行其他小型 RTOS。用来管理这些虚拟机的 RVM 基础设施包括五个守护进程：

安全守护进程（Safety Daemon，SFTD）、

时间守护进程（Timer Daemon，TIMD）、

向量守护进程（Vector Daemon，VCTD）、

超调用守护进程（Virtual Machine Monitor Daemon，VMMD）和

启动守护进程（Initialization Daemon，INITD）。

---

[1] 如 Intel VT-x、AMD AMD-V 等

[2] 如 Intel VT-d 等

同时，每一个 RVM 虚拟机都具备一个虚拟机中断向量处理线程和一个虚拟机用户线程用以处理虚拟机内部事务。

我们推荐将任何稍稍复杂的功能，尤其是那些语言虚拟机、图形界面等放置在虚拟机中实现。这不仅是因为虚拟机提供了一系列库、可能的 POSIX 支持和已认证的软件，更是因为虚拟机在发生故障后会自动重启，而不必重启整个系统。这对于功能安全是至关重要的。

1.4 手册总览

本手册从用户的角度解释了 RVM 的基本组成与运行原理。在本手册中，所有的 typedef 类型之前的“rvm_”前缀均被省去。

1.5 本章参考文献

[1] P. Patel, M. Vanga, and B. B. Brandenburg, "TimerShield: Protecting High-Priority Tasks from Low-Priority Timer Interference," in Real-Time and Embedded Technology and Applications Symposium (RTAS), 2017 IEEE, 2017, pp. 3-12.

[2] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki-a lightweight and flexible operating system for tiny networked sensors," in Local Computer Networks, 2004. 29th Annual IEEE International Conference on, 2004, pp. 455-462.

[3] S. Popov, "The Tangle. url: https://iota. org," IOTA _ Whitepaper. pdf, 2017.

第 2 章 RME 原生进程

2.1 简介

RME 的原生进程直接运行在 RME 之上，RVM 仅仅负责启动它们。RME 原生进程仅仅能使用 RME 本身的系统调用，不能使用虚拟机或者 POSIX 兼容功能，因此其运行速度最快，开销最小。

RME 的系统调用是通过软件中断指令执行的，因此需要一些汇编代码才能访问 RME 的系统调用。由于在实际应用程序中使用汇编代码过于麻烦，因此 RVM 提供了用户态库对于这一部分做了封装，向外提供了这些系统调用的 C 语言接口。这些接口对用户态都是无副作用的，并且是线程安全的。

RVM 的虚拟机用户一般不需要直接调用如下函数中的任何部分，而仅需要关注第三章所述的虚拟化接口部分即可。如果要进行 RVM 的定制，使用这些函数才是必须的。关于 2.4 所提到的系统调用的返回值和参数的具体意义请参见 RME 本身的手册。

我们推荐使用 GSC 编译器生成原生进程工程。它会自动把需要的文件都添加进工程中。

2.2 登记原生进程到 RVM

要登记原生进程到 RVM，需要用户手动使用 RME 系统调用创建其权能表、页表，以及其内部可能使用到的所有内核对象，并将进程内部可以使用的内核对象的权能全部传递到该进程的权能表内部。当进程十分复杂时，这是极度困难和易出错的。GSC 工程编译器可以自动帮助用户完成上述流程，大大降低用户的负担。如果用户执意手动完成该过程，那么需要定义如下两个钩子函数，并在其内部分别创建和初始化各个内核对象。

2.2.1 内核对象创建钩子

用户应当在本钩子内部创建所有的内核对象。

表 2-1 内核对象创建钩子

函数原型	void RVM_Boot_Kobj_Crt(void)
返回值	无。
参数	无。

2.2.2 内核对象初始化钩子

用户应当在本钩子内部初始化所有的内核对象。

表 2-2 内核对象初始化钩子

函数原型	void void RVM_Boot_Kobj_Init(void)
返回值	无。
参数	无。

2.3 原生进程内应当存在的宏定义

原生进程内部需要存在如下的配置宏定义，它们被原生进程内部的其他文件引用。如果使用 GSC 编译器生成虚拟机工程，那么这些宏定义由 GSC 负责填充，否则就要用户手动填充。这些宏定义的列表如下。

表 2-3 原生进程内应当存在的宏定义一览

宏名称	作用
RVM_MPU_REGIONS	微控制器的内存保护单元的区域数。它应当与 RME 中的相关宏维持一致。RME 中相关配置宏名一般是 RME_XXX_MPU_REGIONS，其中 XXX 是架构名称。 例子： #define RVM_MPU_REGIONS (8)
RVM_KMEM_SLOT_ORDER	内核内存分配粒度级数。它应当与 RME 中的同名配置宏维持一致。 例子： #define RVM_KMEM_SLOT_ORDER (4)

2.4 原生进程可以使用的系统调用

RME 提供的所有系统调用都可以在原生进程中被直接使用。这些系统调用的实现的通用部分都位于 Guest 文件夹下的 rmv_guest.c 和 rvm_guest.h 中，架构相关部分则位于子文件夹的 rvm_guest_xxx.c，rvm_guest_xxx.h，rvm_guest_xxx_conf.h 中。

2.4.1 权能表相关系统调用

这些接口向外提供了这些权能表相关调用的 C 语言接口。具体的函数列表如下所述：

表 2-4 权能表相关系统调用

名称	接口函数原型
创建权能表	ret_t RVM_Captbl_Crt(cid_t Cap_Captbl_Crt, cid_t Cap_Kmem, cid_t Cap_Captbl, ptr_t Raddr, ptr_t Entry_Num)
删除权能表	ret_t RVM_Captbl_Del(cid_t Cap_Captbl_Del, cid_t Cap_Del)
传递普通权能	ret_t RVM_Captbl_Add(cid_t Cap_Captbl_Dst, cid_t Cap_Dst, cid_t Cap_Captbl_Src, cid_t Cap_Src, ptr_t Flags) 备注：该函数只能用来传递页目录权能、内核功能调用权能、内核内存权能之外的权能。
传递页表权能	ret_t RVM_Captbl_Pgtbl(cid_t Cap_Captbl_Dst, cid_t Cap_Dst,

名称	接口函数原型
	cid_t Cap_Captbl_Src, cid_t Cap_Src, ptr_t Start, ptr_t End, ptr_t Flags) 备注：该函数不能被用来传递其他类型的权能。Start 和 End 两个参数标志了新的页表权能的起始槽位和终止槽位，也即其操作范围。
传递内核功能调用权能	ret_t RVM_Captbl_Kern(cid_t Cap_Captbl_Dst, cid_t Cap_Dst, cid_t Cap_Captbl_Src, cid_t Cap_Src, ptr_t Start, ptr_t End) 备注：该函数不能被用来传递其他类型的权能。Start 和 End 两个参数标志了新的内核功能调用权能的起始槽位和终止槽位，也即其操作范围。
传递内核内存权能	ret_t RVM_Captbl_Kmem(cid_t Cap_Captbl_Dst, cid_t Cap_Dst, cid_t Cap_Captbl_Src, cid_t Cap_Src, ptr_t Start, ptr_t End, ptr_t Flags) 备注：该函数不能被用来传递其他类型的权能。Start 和 End 两个参数标志了新的内核内存权能的起始地址和终止地址，它们被强制对齐到 64 Byte。
权能冻结	ret_t RVM_Captbl_Frz(cid_t Cap_Captbl_Frz, cid_t Cap_Frz)
权能移除	ret_t RVM_Captbl_Rem(cid_t Cap_Captbl_Rem, cid_t Cap_Rem)

2.4.2 内核功能相关系统调用

这些接口向外提供了这些内核功能相关调用的 C 语言接口。具体的函数列表如下所述：

表 2-5 内核功能相关系统调用

名称	接口函数原型
内核调用激活	ret_t RVM_Kern_Act(cid_t Cap_Kern, ptr_t Func_ID, ptr_t Sub_ID, ptr_t Param1, ptr_t Param2)

2.4.3 页表相关系统调用

这些接口向外提供了这些页表相关调用的 C 语言接口。具体的函数列表如下所述：

表 2-6 页表相关系统调用

名称	接口函数原型
页目录创建	ret_t RVM_Pgtbl_Crt(cid_t Cap_Captbl, cid_t Cap_Kmem, cid_t Cap_Pgtbl, ptr_t Raddr, ptr_t Start_Addr, ptr_t Top_Flag, ptr_t Size_Order, ptr_t Num_Order)

名称	接口函数原型
删除页目录	ret_t RVM_Pgtbl_Del(cid_t Cap_Captbl, cid_t Cap_Pgtbl)
映射内存页	ret_t RVM_Pgtbl_Add(cid_t Cap_Pgtbl_Dst, ptr_t Pos_Dst, ptr_t Flags_Dst, cid_t Cap_Pgtbl_Src, ptr_t Pos_Src, ptr_t Index)
移除内存页	ret_t RVM_Pgtbl_Rem(cid_t Cap_Pgtbl, ptr_t Pos)
构造页目录	ret_t RVM_Pgtbl_Con(cid_t Cap_Pgtbl_Parent, ptr_t Pos, cid_t Cap_Pgtbl_Child, ptr_t Flags_Childs)
析构页目录	ret_t RVM_Pgtbl_Des(cid_t Cap_Pgtbl_Parent, ptr_t Pos, cid_t Cap_Pgtbl_Child)

2.4.4 进程相关系统调用

这些接口向外提供了这些进程相关调用的 C 语言接口。具体的函数列表如下所述：

表 2-7 进程相关系统调用

名称	接口函数原型
创建进程	ret_t RVM_Proc_Crt(cid_t Cap_Captbl_Crt, cid_t Cap_Proc, cid_t Cap_Captbl, cid_t Cap_Pgtbl)
删除进程	ret_t RVM_Proc_Del(cid_t Cap_Captbl, cid_t Cap_Proc)
更改权能表	ret_t RVM_Proc_Cpt(cid_t Cap_Proc, cid_t Cap_Captbl)
更改页表	ret_t RVM_Proc_Pgt(cid_t Cap_Proc, cid_t Cap_Pgtbl)

2.4.5 线程相关系统调用

这些接口向外提供了这些线程相关调用的 C 语言接口。具体的函数列表如下所述：

表 2-8 线程相关系统调用

名称	接口函数原型
创建线程	ret_t RVM_Thd_Crt(cid_t Cap_Captbl, cid_t Cap_Kmem, cid_t Cap_Thd, cid_t Cap_Proc, ptr_t Max_Prio, ptr_t Raddr)
删除线程	ret_t RVM_Thd_Del(cid_t Cap_Captbl, cid_t Cap_Thd)
设置执行属性	ret_t RVM_Thd_Exec_Set(cid_t Cap_Thd, ptr_t Entry, ptr_t Param, ptr_t Stack)
设置虚拟属性	ret_t RVM_Thd_Hyp_Set(cid_t Cap_Thd, ptr_t Kaddr)
线程绑定	ret_t RVM_Thd_Sched_Bind(cid_t Cap_Thd, cid_t Cap_Thd_Sched, cid_t Cap_Sig, tid_t TID, ptr_t Prio)
更改优先级	ret_t RVM_Thd_Sched_Prio(cid_t Cap_Thd, ptr_t Prio)

名称	接口函数原型
接收调度事件	ret_t RVM_Thd_Sched_Rcv(cid_t Cap_Thd, ptr_t* Fault)
解除绑定	ret_t RVM_Thd_Sched_Free(cid_t Cap_Thd)
传递时间片	ret_t RVM_Thd_Time_Xfer(cid_t Cap_Thd_Dst, cid_t Cap_Thd_Src, ptr_t Time)
切换到某线程	ret_t RVM_Thd_Swt(cid_t Cap_Thd, ptr_t Full_Yield)

2.4.6 异步信号端点相关系统调用

这些接口向外提供了这些信号端点相关调用的 C 语言接口。具体的函数列表如下所述：

表 2-9 异步信号端点相关系统调用

名称	接口函数原型
创建信号端点	ret_t RVM_Sig_Crt(cid_t Cap_Captbl, cid_t Cap_Sig)
删除信号端点	ret_t RVM_Sig_Del(cid_t Cap_Captbl, cid_t Cap_Sig)
向端点发送	ret_t RVM_Sig_Snd(cid_t Cap_Sig)
从端点接收	ret_t RVM_Sig_Rcv(cid_t Cap_Sig, ptr_t Option)

2.4.7 同步迁移调用相关系统调用

这些接口向外提供了这些线程迁移相关调用的 C 语言接口。具体的函数列表如下所述：

表 2-10 同步迁移调用相关系统调用

名称	接口函数原型
创建同步迁移调用	ret_t RVM_Inv_Crt(cid_t Cap_Captbl, cid_t Cap_Kmem, cid_t Cap_Inv, cid_t Cap_Proc, ptr_t Raddr)
删除同步迁移调用	ret_t RVM_Inv_Del(cid_t Cap_Captbl, cid_t Cap_Inv)
设置执行属性	ret_t RVM_Inv_Set(cid_t Cap_Inv, ptr_t Entry, ptr_t Stack, ptr_t Fault_Ret_Flag)
激活同步迁移调用	ret_t RVM_Inv_Act(cid_t Cap_Inv, ptr_t Param, ptr_t* Retval)
从同步迁移调用返回	ret_t RVM_Inv_Ret(ptr_t Retval)

2.5 原生进程可以使用的其他助手函数

为了方便常用数据结构编写、调试打印、线程栈初始化，RVM 还提供了一些助手函数。下面列出的所有助手函数均可以在原生进程中使用。

表 2-11 原生进程可以使用的其他助手函数一览

函数	意义
RVM_Clear	变量清空。
RVM_List_Crt	创建双向循环链表。
RVM_List_Del	在双向循环链表中删除节点。
RVM_List_Ins	在双向循环链表中插入节点。
RVM_Putchar	打印单个字符。
RVM_Print_Int	打印整形数字。
RVM_Print_Uint	打印无符号整形数字。
RVM_Print_String	打印字符串。

2.5.1 变量清空

该函数用来在内核中清零一片区域。该函数实质上等价于 C 语言运行时库的 memset 函数填充 0 时的特殊情况。

表 2-12 变量清空的所需参数

原型	void RVM_Clear(void* Addr, ptr_t Size)	
参数名称	类型	描述
Addr	void*	需要清零区域的起始地址。
Size	ptr_t	需要清零区域的字节数。

2.5.2 创建双向循环链表

本操作初始化双向循环链表的链表头。

表 2-13 创建双向循环链表

函数原型	void RVM_List_Crt(volatile struct RVM_List* Head)
返回值	无。
参数	volatile struct RVM_List* Head 指向要初始化的链表头结构体的指针。

2.5.3 在双向循环链表中删除节点

本操作从双向链表中删除一个或一系列节点。

表 2-14 在双向循环链表中删除节点

函数原型	void RVM_List_Del(volatile struct RVM_List* Prev, volatile struct RVM_List* Next)
返回值	无。
参数	volatile struct RVM_List* Prev 指向要删除的节点（组）的前继节点的指针。
	volatile struct RVM_List* Next 指向要删除的节点（组）的后继节点的指针。

2.5.4 在双向循环链表中插入节点

本操作从双向链表中插入一个节点。

表 2-15 在双向循环链表中插入节点

函数原型	void RVM_List_Ins(volatile struct RVM_List* New, volatile struct RVM_List* Prev, volatile struct RVM_List* Next)
返回值	无。
参数	volatile struct RVM_List* New 指向要插入的新节点的指针。
	volatile struct RVM_List* Prev 指向要被插入的位置的前继节点的指针。
	volatile struct RVM_List* Next 指向要被插入的位置的后继节点的指针。

2.5.5 打印单个字符

本操作打印一个字符到控制台。该函数需要用户自行声明和实现。在该函数的实现中，通常只需要重定向其输出到某外设即可。

表 2-16 打印单个字符

函数原型	ptr_t RVM_Putchar(char Char)
意义	输出一个字符到控制台。
返回值	ptr_t 总是返回 0。
参数	char Char 要输出到系统控制台的字符。

2.5.6 打印整形数字

本操作以包含符号的十进制打印一个机器字长的整形数字到调试控制台。要使用这个函数，需要 RVM_Putchar 在进程内被声明和实现。

表 2-17 打印整形数字

函数原型	cnt_t RVM_Print_Int(cnt_t Int)
返回值	cnt_t 返回打印的字符数量。
参数	cnt_t Int 要打印的整形数字。

2.5.7 打印无符号整形数字

本操作以无前缀十六进制打印一个机器字长的无符号整形数字到调试控制台。要使用这个函数，需要 RVM_Putchar 在进程内被声明和实现。

表 2-18 打印无符号整形数字

函数原型	cnt_t RVM_Print_Uint(ptr_t Uint)
返回值	cnt_t 返回打印的字符数量。
参数	ptr_t Uint 要打印的无符号整形数字。

2.5.8 打印字符串

本操作打印一个最长不超过 127 字符的字符串到调试控制台。要使用这个函数，需要 RVM_Putchar 在进程内被声明和实现。

表 2-19 打印字符串

函数原型	cnt_t RVM_Print_String(s8_t* String)
返回值	cnt_t 返回打印的字符数量。
参数	s8_t* String 要打印的字符串。

2.6 本章参考文献

无

第 3 章 RVM 虚拟机进程

3.1 简介

RVM 虚拟机运行在一个基于准虚拟化技术实现的虚拟机监视器上。它实现了虚拟化的所有基本功能：内存管理、CPU 管理、中断向量虚拟化和虚拟机错误处理。关于这些实现的具体信息请参见下文所述。虚拟化的总架构见下图。图中实现代表硬件隔离机制，虚线代表软件模组的边界。

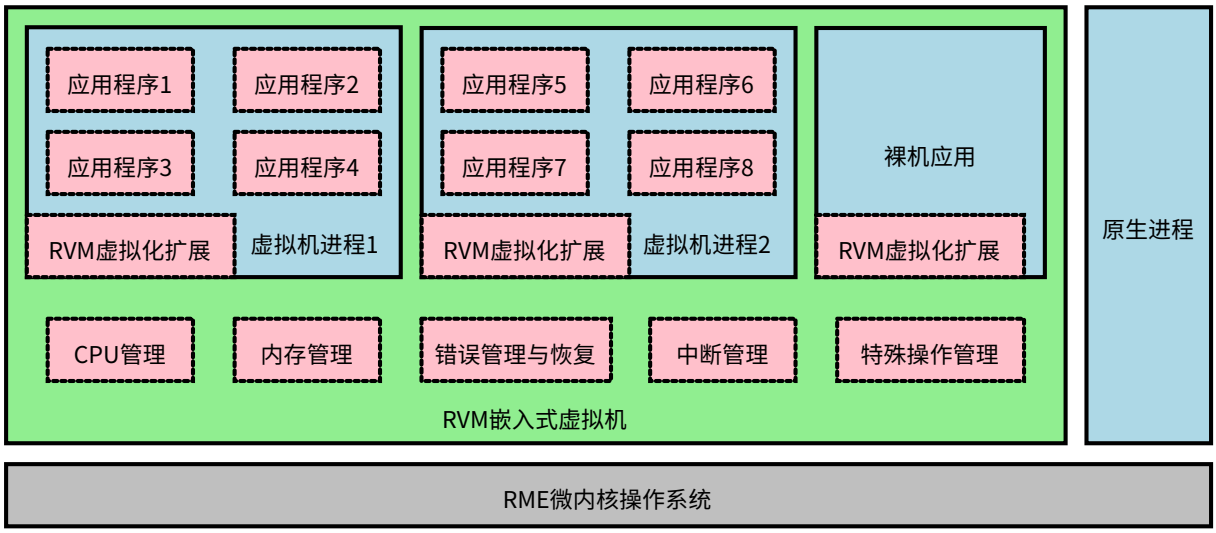

图 3-1 RVM 架构简图

在 RVM 内部，各个虚拟机也有它们各自的优先级。这个优先级是静态指定的。不同优先级的虚拟机之间采取抢占式调度策略，相同优先级的虚拟机之间采取时间片轮转调度策略。

3.2 虚拟化架构详述

第一个启动执行的是启动守护进程。RVM 在它开始执行后会加载各个虚拟机。然后，四个守护进程会启动，并且负责对虚拟机提供服务。这四个守护进程分别是安全守护进程，向量守护进程，超调用守护进程和时间守护进程。其中，安全守护进程会在虚拟机发生错误时直接重启虚拟机本身[1]；向量守护进程会把来自设备的中断重定向给对应虚拟机的虚拟中断向量；超调用守护进程负责处理超级调用和虚拟机调度；时间守护进程则负责向各个虚拟机发送时钟中断。

每一个虚拟机内部则运行两个线程，一个负责处理虚拟机内部的中断，另一个则负责运行用户代码。具体的虚拟化原理会在以下各小节分别介绍。

3.2.1 优先级配置

在 RVM 的五个守护进程中，错误处理守护进程永远运行在系统的最高优先级。紧随其后的是 RVM 中的用户定义快速中断处理线程，它们负责快速处理系统中的部分来不及发送给虚拟机处理的中断。再

[1] RVM 不支持用户自定义错误处理

接下来是其他三个守护进程，它们的优先级要求比虚拟机线程高。然后则是在运行的虚拟机的中断处理线程和用户线程，其中中断处理线程的优先级又要高于用户处理线程。之后是系统本身的 Init 进程，该进程负责无限循环调用空闲钩子，在没有虚拟机就绪时让系统进入休眠。最后则是未在运行的虚拟机的中断处理线程和用户程序线程。整个虚拟化体系的各线程的中断优先级如下所示：

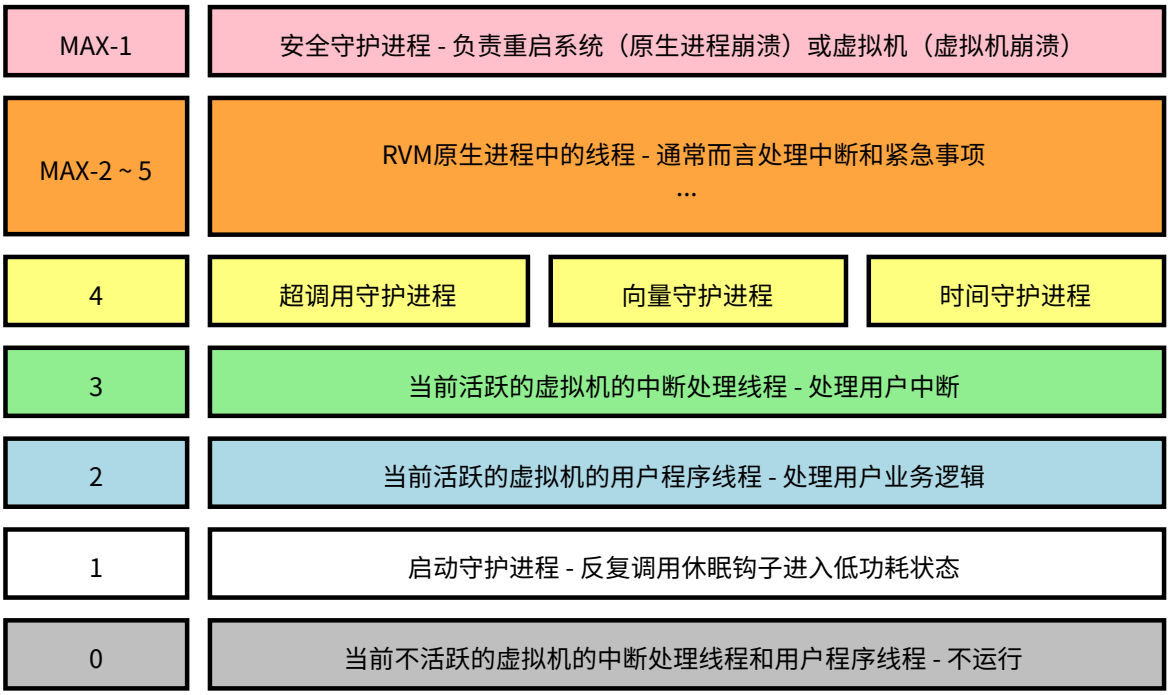

图 3-2 虚拟化体系的中断优先级

各个虚拟机采用固定优先级时间片轮转法调度。它们之间的优先级则是由调度器守护进程决定的。需要注意的是，为了系统安全，虚拟机的时间片和优先级是在系统编译时静态决定好的，RVM 并不提供可以修改这两点的超级调用。这从根本上保证了虚拟机之间的时间隔离，使得没有虚拟机可以破坏整个系统的实时性。

3.2.2 内存管理

在虚拟机中，最重要的事项之一就是内存管理。考虑到微控制器应用的一般特点，每个虚拟机的内存，包括私有内存和共享内存，都是在运行之前由用户静态指定的。在虚拟机开始运行之后，不允许再新申请内存。对于每一种处理器，页表信息的格式都是不同的。请参看相应处理器的使用手册确定该处理器的页表形式。

3.2.3 CPU 的虚拟化

RVM 采用双线程的方法进行 CPU 虚拟化。一个线程负责中断管理，另外一个线程则负责处理用户程序。负责中断处理的线程平时不运行，只有当接收到了来自 RVM 的中断信号时才运行。当中断处理完成后，中断处理线程又阻塞，用户程序线程开始运行。

如果需要在中断线程中修改用户线程的寄存器组，由于用户线程的寄存器组在它被切换出时会被保存在某个可指定的地址，因此只要修改那个地址上的寄存器数据即可。

3.2.4 中断的虚拟化

对于虚拟机而言，中断虚拟化是必不可少的。RVM 的每个客户机中都具有一个专门处理中断的线程，该线程的优先级比处理用户程序的线程优先级高。该线程平时阻塞在虚拟中断向量端点上，该端点在每个虚拟机内部都位于权能号 1 的位置，直到收到中断信号才开始处理。

系统中的中断大概可以分成三类，第一类是时钟中断，它负责向操作系统提供时钟嘀嗒；第二类是线程切换中断，它负责进行 RTOS 的线程切换；第三类则是其他系统中断，它们负责响应其他外设的事务。这三类中断的特点也不同：时钟中断是周期性频繁发生的；系统调用中断本质上是自己发送给自己的软中断；其他系统中断则是不时发生的无规律中断。因此，RVM 对于这三种中断采取了不同的虚拟化策略。

3.2.4.1 时钟中断的虚拟化

时钟中断由 RME 的底层定时器产生，并且通过内核端点的方式发送到 RVM 的时钟处理守护线程。该守护线程会计算当前经过的时间周期数，并且按时给各个虚拟机发送时钟中断。各个虚拟机的中断处理线程收到时钟中断后负责处理自身内部的定时器，并且调度其线程。

3.2.4.2 线程切换中断的虚拟化

线程切换中断本质上是一个软中断，它来源于虚拟机自身的内部。这个中断将直接被送往虚拟机的中断线程进行处理，不会经过 RVM。

虚拟机收到这个中断后，由虚拟中断处理线程调用用户注册的线程切换钩子。在该钩子中，用户保存和恢复各个用户程序处理线程的各个寄存器，完成用户处理线程对应的 RTOS 线程的切换。保存和恢复的一般手段是压栈弹栈，这一点和一般 RTOS 并无差异。差异只有两点：第一点是，压栈弹栈现在可以由 C 语言写成[1]，第二点是压栈弹栈也要包括用户线程使用的超级调用参数缓冲区，它位于 struct RVM_Param_Area 结构体的 struct RVM_Param User 域。关于第二点，我们在 3.2.5 中将详细介绍。

3.2.4.3 其他中断的虚拟化

由于不同的虚拟机需要接收不同的外设中断，因此其他外设中断要按需送往各个虚拟机。因此，系统提供了调用接口来让各个虚拟机设置自己的每个虚拟中断向量的物理中断源。每个虚拟机能且只能设置自己的虚拟中断向量对应的物理中断源。一旦该物理中断被触发，登记于这个物理中断上的所有的虚拟机的相应虚拟中断向量均会收到此中断。

3.2.5 超级调用的设计

[1] 具体例子可以参见 RMP 的 RVM 移植

虚拟机可以通过超级调用向虚拟机监视器请求服务。超级调用的设计原理是，由虚拟机先将各个参数写到与 RVM 约定好的超调用参数缓冲区，然后向 RVM 的超调用守护进程的信号端点发送信号。为了达到这点，RVM 的超调用守护进程的信号端点权能必须被传递至虚拟机内部，并位于权能号 0 的位置。

由于多个线程同时调用超级调用可能导致在超调用参数缓冲区出现竞争冒险，因此 RVM 设定了特别的机制来保证竞争冒险不会出现。首先，在中断中处理的超级调用使用自己的单独的缓冲区，因此中断中调用的超级调用不会和用户线程的超级调用产生竞争冒险；其次，在 RTOS 的移植中，用户线程使用的参数缓冲区将和用户线程寄存器组一起压栈弹栈，这确保了用户线程切换时不会导致竞争冒险。

3.3 登记虚拟机进程到 RVM

在 RVM 中，每个虚拟机的配置被表示成一个 struct RVM_Virt_Map 类型的结构体。这些结构体组成一个常量结构体数组 RVM_Virt_Map_DB，内部储存有关于系统中全部虚拟机的信息。这些信息包括虚拟机的起始执行地址、起始栈地址、中断标志数组地址、参数传递区地址、所使用的内存区域和所使用的特殊内核调用等等。这些信息帮助 RVM 正确加载和配置各个虚拟机。此外，RVM 还需要存放这些虚拟机状态的另一个结构体类型 struct RVM_Virt 组成的数组 RVM_Virt_DB 来存放那些虚拟机运行状态的有关数据。这两个结构体数组的长度必须是一样的。用户需要在 RVM 启动时调用函数 RVM_Virt_Crt 批量注册这些虚拟机到系统中。除了结构体信息填充以外，还要进行一些内核对象的创建和初始化。

GSC 可以完全自动化地进行这个过程，因此我们推荐使用 GSC 建立各个虚拟机。

3.3.1 虚拟机信息的填充

用户应当静态声明结构体数组 const struct RVM_Virt_Map RVM_Virt_Map_DB[]，并填充各个虚拟机的相关内容。该结构体的各个域的意义如下表所列。

表 3-1 struct RVM_Virt_Map 中各个域的意义

域名称	作用
s8_t* Name	虚拟机的名称。它是一个字符串。它实际上并不会被 RVM 使用，仅作为方便用户调试之用。
ptr_t Prio	虚拟机的优先级。这个优先级必须小于 RVM 配置时的虚拟机优先级数。
ptr_t Slices	虚拟机的运行时间片。该值不可以设置为 0。
ptr_t Period	虚拟机时钟中断的周期。设置为 X，即表示 RVM 在收到 X 个时钟嘀嗒后会给该虚拟机发送一个虚拟时钟中断。推荐将该值设置为一个大于 10 的数字。
ptr_t Watchdog	虚拟机看门狗超时时间。设置为 X，即表示该虚拟机运行超过 X 个时钟嘀嗒后还没有喂狗则重置该虚拟机。如果该项设置为 0，则表明未对此虚拟机启用看门狗。
ptr_t Vect_Num	虚拟机虚拟中断向量的数目，这个数目不能超过 1920。通常而言任何一

域名称	作用
	个虚拟机都很难用到如此大量的中断源。
struct RVM_Regs* Reg_Base	虚拟机的虚拟寄存器缓冲区地址。它指向的内存地址必须不小于本架构的 struct RVM_Regs 结构体的大小。它应当与登记在虚拟机进程内部的 RVM_VIRT_REG_BASE 配置项维持一致。
struct RVM_Param_Area* Param_Base	虚拟机的超调用参数缓冲区地址。它指向的内存地址必须不小于 struct RVM_Param_Area 结构体的大小。它应当与登记在虚拟机进程内部的 RVM_VIRT_PARAM_BASE 配置项维持一致。
struct RVM_Vect_Flag* Vctf_Base	虚拟机的虚拟中断向量标志位区地址。它指向的内存地址必须不小于 256 字节。它应当与登记在虚拟机进程内部的 RVM_VIRT_VCTF_BASE 配置项维持一致。
ptr_t Entry_Code_Front	虚拟机的跳转用代码片段的地址。它只在某些运行前需要初始化栈的架构上存在和使用。
cid_t Vect_Sig_Cap	虚拟机中断向量端点的权能号。这个权能号是 RVM 中的权能号。下面提到的各个权能号也是一样的。
cid_t Vect_Thd_Cap;	虚拟机中断向量处理线程的权能号。
tid_t Vect_TID	虚拟机中断向量处理线程的线程 ID。推荐填充为和其权能号一样的值。
ptr_t Vect_Entry	虚拟机中断向量处理线程的入口点。
ptr_t Vect_Stack_Base	虚拟机中断向量处理线程的栈基址。
ptr_t Vect_Stack_Size	虚拟机中断向量处理线程的栈大小。这个大小必须超过使用栈量最大的虚拟中断向量函数的栈大小。
cid_t User_Thd_Cap	虚拟机用户程序处理线程的权能号。
tid_t User_TID	虚拟机用户程序处理线程的线程 ID。推荐填充为和其权能号一样的值。
ptr_t User_Entry	虚拟机用户程序处理线程的入口点。
ptr_t User_Stack_Base	虚拟机用户程序处理线程的栈基址。
ptr_t User_Stack_Size	虚拟机用户程序处理线程的栈大小。对于单线程裸机程序，这个大小必须超过用户程序逻辑使用的栈大小；由于 RTOS 在启动后的各个线程往往会切换到由 RTOS 自行管理的专用栈，因此该值只要超过 RTOS 启动和初始化序列使用的栈大小即可。

3.3.2 创建虚拟机中的内核对象并初始化

在完成虚拟机信息填充后，我们还要为每个虚拟机创建一个信号端点，并将其传递到该虚拟机内部权能表的 1 号位，使中断线程处理程序能进行中断接收。该被传递的权能应当具备所有的接收和发送方式的操作属性，也即 RVM_SIG_FLAG_ALL。另外，还要将虚拟机超级调用的信号端点传递到该虚拟机内

部权能表的 0 号位，使虚拟机能够发起超级调用。该被传递的权能应当仅允许发送操作属性，也即 RVM_SIG_FLAG_SND。

3.3.3 虚拟机批量注册

用户首先应当静态声明结构体数组 struct RVM_Virt RVM_Virt_DB[]以存放运行时数据，然后在内核对象初始化钩子 RVM_Boot_Kobj_Init 的最后调用函数 RVM_Virt_Crt 批量注册虚拟机。

表 3-2 虚拟机批量注册钩子

函数原型	void RVM_Virt_Crt(struct RVM_Virt* Virt_DB, struct RVM_Virt_Map* Virt_Map, rvm_ptr_t Virt_Size)
返回值	无。
参数	struct RVM_Virt* Virt_DB 静态声明的虚拟机运行时数据结构体数组的首地址。
	struct RVM_Virt_Map* Virt_Map 静态声明的虚拟机信息结构体数组的首地址。
	ptr_t Virt_Size 虚拟机的数量。静态声明的上述两个结构体数组的长度均应该等于这个值。

3.4 虚拟机进程内部应当存在的宏定义

虚拟机内部需要存在如下的配置宏定义。如果使用 GSC 编译器生成虚拟机工程，那么这些宏定义由 GSC 负责填充，否则就要用户手动填充。这些宏定义的列表如下。

表 3-3 虚拟机进程内应当存在的宏定义一览

宏名称	作用
RVM_MPU_REGIONS	微控制器的内存保护单元的区域数。它应当与 RME 中的相关宏维持一致。RME 中相关配置宏名一般是 RME_XXX_MPU_REGIONS，其中 XXX 是架构名称。 例子： #define RVM_MPU_REGIONS (8)
RVM_KMEM_SLOT_ORDER	内核内存分配粒度级数。它应当与 RME 中的同名配置宏维持一致。 例子： #define RVM_KMEM_SLOT_ORDER (4)
RVM_VIRT_VECT_NUM	虚拟中断向量的数量。它应当与登记在 RVM 虚拟机结构体中的同

宏名称	作用
	名配置项维持一致。 例子： #define RVM_VIRT_VECT_NUM (10)
RVM_VIRT_VCTF_BASE	虚拟中断向量标志位的基地址。它应当与登记在 RVM 虚拟机结构体中的同名配置项维持一致。 例子： #define RVM_VIRT_VCTF_BASE (0x20005700)
RVM_VIRT_REG_BASE	用户线程虚拟寄存器组的基地址。它应当与登记在 RVM 中的同名配置项维持一致。 例子： #define RVM_VIRT_REG_BASE (0x20005400)
RVM_VIRT_PARAM_BASE	超调用参数缓冲区的基地址。它应当与登记在 RVM 中的同名配置项维持一致。 例子： #define RVM_VIRT_PARAM_BASE (0x20005600)

3.5 虚拟机可以使用的超级调用

虚拟机间的通信和基本管理都是通过超级调用实现的。这些超级调用可以在用户线程中调用，也可以在虚拟中断服务程序中调用。

表 3-4 虚拟机可以使用的内部功能函数一览

函数	意义
RVM_Hyp_Ena_Int	开启虚拟总中断。
RVM_Hyp_Dis_Int	关闭虚拟总中断。
RVM_Hyp_Reg_Phys	映射物理中断源到虚拟中断源。
RVM_Hyp_Reg_Evt	映射事件源到虚拟中断源。
RVM_Hyp_Del_Vect	解除虚拟中断源上的映射。
RVM_Hyp_Add_Evt	授权向某事件源进行发送。
RVM_Hyp_Del_Evt	撤销向某事件源进行发送的授权。
RVM_Hyp_Lock_Vect	锁定虚拟中断源映射。
RVM_Hyp_Send_Evt	向某事件源进行发送。
RVM_Hyp_Wait_Vect	等待虚拟中断源被触发。

函数	意义
RVM_Hyp_Feed_Wdog	激活并投喂看门狗。

3.5.1 开启虚拟总中断

本超级调用开启虚拟机的虚拟中断接收。虚拟机在启动后，虚拟中断默认是关闭的，此时不会再有更多中断被发送到虚拟机。这是为了模拟微控制器的行为：微控制器启动时其总中断一般也是关闭的。因此，在虚拟机中的 RTOS 需要在初始化结束后调用这一函数开启虚拟中断。本超级调用没有嵌套计数，因此需要用户自行进行嵌套计数。

表 3-5 开启虚拟总中断

函数原型	void RVM_Hyp_Ena_Int(void)
返回值	无。该调用总是成功。
参数	无。

3.5.2 关闭虚拟总中断

本超级调用关闭虚拟中断。当虚拟中断被关闭后，RVM 将不再向此虚拟机发送任何中断。本超级调用没有嵌套计数，因此需要用户自行进行嵌套计数。

表 3-6 关闭虚拟总中断

函数原型	void RVM_Hyp_Dis_Int(void)
返回值	无。该调用总是成功。
参数	无。

3.5.3 映射物理中断源到虚拟中断源

本超级调用映射一个物理中断源到本虚拟机的一个虚拟中断源。在物理中断源被触发后，对应的虚拟中断源上会收到中断，虚拟机将调用注册于该中断的处理函数。一个物理中断源只能被映射到一个虚拟机一次，而一个虚拟中断源只能被一个物理中断源或事件源映射到。每一次映射都会消耗一个映射块，系统中的映射块数量由 RVM 在配置时的 RVM_MAP_NUM 决定。

这一超级调用并非是实时的。因此，我们仅推荐在系统启动时调用它。

表 3-7 映射物理中断源到虚拟中断源

函数原型	ret_t RVM_Hyp_Reg_Phys(ptr_t Phys_Num, ptr_t Vect_Num)
返回值	ret_t 如果成功，返回 0。如果失败则会返回如下负值：

	RVM_ERR_RANGE	传入的物理中断向量号或虚拟中断向量号越界。
	RVM_ERR_STATE	本虚拟机的虚拟中断源映射关系已经锁定。
	RVM_ERR_PHYS	本虚拟机已经注册了该物理向量到某个虚拟中断向量。
	RVM_ERR_VIRT	本虚拟中断向量已被注册到了物理中断向量或者事件源。
	RVM_ERR_MAP	映射块用尽，无法进行映射。
参数	ptr_t Phys_Num 物理中断源的中断向量号。	
	ptr_t Vect_Num 虚拟中断源的中断向量号。	

3.5.4 映射事件源到虚拟中断源

本超级调用映射一个事件源到本虚拟机的一个虚拟中断源。在事件源被触发后，对应的虚拟中断源上会收到中断，虚拟机将调用注册于该中断的处理函数。一个事件源只能被映射到一个虚拟机一次，而一个虚拟中断源只能被一个物理中断源或事件源映射到。每一次映射都会消耗一个映射块，系统中的映射块数量由 RVM 在配置时的 RVM_MAP_NUM 决定。

这一超级调用并非是实时的。因此，我们仅推荐在系统启动时调用它。

表 3-8 映射事件源到虚拟中断源

函数原型	ret_t RVM_Hyp_Reg_Evt(ptr_t Evt_Num, ptr_t Vect_Num)	
返回值	ret_t 如果成功，返回 0。如果失败则会返回如下负值：	
	RVM_ERR_RANGE	传入的事件源号或虚拟中断向量号越界。
	RVM_ERR_STATE	本虚拟机的虚拟中断源映射关系已经锁定。
	RVM_ERR_PHYS	本虚拟机已经注册了该事件源到某个虚拟中断向量。
	RVM_ERR_VIRT	本虚拟中断向量已被注册到了物理中断向量或者事件源。
	RVM_ERR_MAP	映射块用尽，无法进行映射。
参数	ptr_t Evt_Num 事件源的事件源号。	
	ptr_t Vect_Num 虚拟中断源的中断向量号。	

3.5.5 解除虚拟中断源上的映射

本超级调用解除虚拟中断源上的任何映射，无论是物理中断源还是事件源。

表 3-9 解除虚拟中断源上的映射

函数原型	ret_t RVM_Hyp_Del_Vect(ptr_t Vect_Num)	
返回值	ret_t 如果成功，返回 0。如果失败则会返回如下负值：	
	RVM_ERR_RANGE	传入的虚拟中断向量号越界。
	RVM_ERR_STATE	本虚拟机的虚拟中断源映射关系已经锁定。
	RVM_ERR_VIRT	本虚拟中断向量没有被注册到任何物理中断向量或者事件源。
参数	ptr_t Vect_Num 虚拟中断源的中断向量号。	

3.5.6 授权向某事件源进行发送

本超级调用授权虚拟机向某个事件源进行发送操作。要向任何一个事件源进行发送操作都需要此等授权。

表 3-10 授权向某事件源进行发送

函数原型	ret_t RVM_Hyp_Add_Evt(ptr_t Evt_Num)	
返回值	ret_t 如果成功，返回 0。如果失败则会返回如下负值：	
	RVM_ERR_RANGE	传入的事件源号越界。
	RVM_ERR_STATE	本虚拟机的虚拟中断源映射关系已经锁定。
	RVM_ERR_EVT	本虚拟机已经具有向该事件源进行发送的权限，无需授权。
参数	ptr_t Evt_Num 事件源的事件源号。	

3.5.7 撤销向某事件源进行发送的授权

本超级调用撤销虚拟机向某个事件源进行发送的授权。撤销后，虚拟机将不再能向该事件源进行发送。

表 3-11 撤销向某事件源进行发送的授权

函数原型	ret_t RVM_Hyp_Del_Evt(ptr_t Evt_Num)
返回值	ret_t

	如果成功，返回 0。如果失败则会返回如下负值：	
	RVM_ERR_RANGE	传入的事件源号越界。
	RVM_ERR_STATE	本虚拟机的虚拟中断源映射关系已经锁定。
	RVM_ERR_EVT	本虚拟机原本就没有向该事件源进行发送的权限，无需撤销。
参数	ptr_t Evt_Num 事件源的事件源号。	

3.5.8 锁定虚拟中断源映射

本超级调用锁定当前虚拟机的虚拟中断源映射，这包括了物理中断源到虚拟中断源的映射、事件源到虚拟中断源的映射和虚拟机的事件源发送授权。锁定一经生效即为永久的。除非当前虚拟机崩溃重启，否则不可能对虚拟中断源映射做出任何进一步的修改。这是防止某些虚拟机被攻破后大量映射物理中断源或事件源，或者向事件源大量发送而造成拒绝服务攻击。

我们推荐在每一个虚拟机的启动时即完成所有上述初始化，然后使用本超级调用锁定它们。由于在微控制器上，修改放置在 Flash 中或在上电时加载到 RAM 中的代码段是不可能的[1]，因此可以保证在锁定前当前虚拟机的执行都是可信的。这样，在锁定后可以完全保证系统的信息安全。只有对那些非常复杂、高度灵活的虚拟机，我们才不进行锁定。

表 3-12 锁定虚拟中断源映射

函数原型	ret_t RVM_Hyp_Lock_Vect(void)	
返回值	ret_t 如果成功，返回 0。如果失败则会返回如下负值：	
	RVM_ERR_STATE	本虚拟机的虚拟中断源映射关系已经锁定，无需再次锁定。
参数	无。	

3.5.9 向某事件源进行发送

本超级调用会令虚拟机向某个事件源进行发送。此发送将会激活所有的在该事件源上注册了的虚拟机的中断向量。

表 3-13 向某事件源进行发送

函数原型	ret_t RVM_Hyp_Send_Evt(ptr_t Evt_Num)
返回值	ret_t 如果成功，返回 0。如果失败则会返回如下负值：

[1] 即便 RME 内核也没有权限修改它们，只有加载器（Bootloader）有此权限

	RVM_ERR_RANGE	传入的事件源号越界。
	RVM_ERR_EVT	本虚拟机没有向该事件源进行发送的权限。
参数	ptr_t Evt_Num 事件源的事件源号。	

3.5.10 等待虚拟中断源被触发

本超级调用会让虚拟机进入休眠状态，等待下一次它的任何虚拟中断源被激活再继续执行。要调用这个超级调用，虚拟机的虚拟总中断必须是开启状态。通常而言，这个超级调用会被放置在 RTOS 的空闲线程钩子内被反复调用，它在微控制器上的等价物是那些可能导致进入休眠状态的指令[1]。

表 3-14 等待虚拟中断源被触发

函数原型	ret_t RVM_Hyp_Wait_Vect(void)	
返回值	ret_t 如果成功，返回 0。如果失败则会返回如下负值：	
	RVM_ERR_STATE	本虚拟机的虚拟总中断处于关闭状态，不能进入休眠。
参数	无。	

3.5.11 激活并投喂看门狗

当虚拟机信息中的 ptr_t Watchdog 项不为 0 时，本超级调用可用。在虚拟机创建时，看门狗默认是关闭的；而当本超级调用第一次被调用时，看门狗即会启动和开始倒数来计算虚拟机的运行时间[2]，一旦倒数到 0 则重启虚拟机。本超级调用也可以用来投喂看门狗。看门狗一经启用便无法停止，除非虚拟机因故障或看门狗超时重启。

表 3-15 激活并投喂看门狗

函数原型	ret_t RVM_Hyp_Feed_Wdog(void)	
返回值	ret_t 如果成功，返回 0。如果失败则会返回如下负值：	
	RVM_ERR_STATE	ptr_t Watchdog 项为 0，本虚拟机无法启用看门狗。
参数	无。	

3.6 虚拟机可以使用的内部功能函数

[1] 如 ARMv7-M 的 WFE 和 WFI

[2] 不是系统的运行时间，看门狗只计算虚拟机本身的运行时间

除了超级调用之外，虚拟机还有一些内部功能函数可以使用。这些内部功能函数不是超级调用，它们仅仅对虚拟机内部的数据结构进行调整来达到它们的效果，因此比超级调用来得高效的多。

表 3-16 虚拟机可以使用的内部功能函数一览

函数	意义
RVM_Virt_Init	虚拟机数据结构初始化。
RVM_Vect_Loop	运行虚拟中断向量处理循环。
RVM_Virt_Mask_Int	屏蔽虚拟中断向量处理。
RVM_Virt_Unmask_Int	解除虚拟中断向量处理屏蔽。
RVM_Virt_Reg_Vect	注册普通虚拟中断向量处理函数。
RVM_Virt_Reg_Timer	注册时钟虚拟中断向量处理函数
RVM_Virt_Reg_Ctxsw	注册线程切换虚拟中断向量处理函数。
RVM_Virt_Yield	触发虚拟线程切换中断向量。

3.6.1 虚拟机数据结构初始化

本操作初始化虚拟机的内部数据结构。该操作应该由虚拟机中断处理向量在虚拟机启动时最先调用。在使用 GSC 生成工程的场合，GSC 会自动生成对本函数的调用，无需用户手动引用本函数。

表 3-17 虚拟机数据结构初始化

函数原型	void RVM_Virt_Init(void)
返回值	无。该操作总是成功。
参数	无。

3.6.2 运行虚拟中断向量处理循环

本操作处理虚拟机内部的中断向量。该操作应该由虚拟机中断处理向量在初始化过数据结构后调用，并且永不返回。在使用 GSC 生成工程的场合，GSC 会自动生成对本函数的调用，无需用户手动引用本函数。

表 3-18 运行虚拟中断向量处理循环

函数原型	void RVM_Vect_Loop(void)
返回值	无。该操作永不返回。
参数	无。

3.6.3 屏蔽虚拟中断向量处理

本操作屏蔽虚拟机内部对中断向量的处理。虚拟机仍然会接收中断，因此用户逻辑线程仍然会被中断处理线程打断，但是接收到的中断都将悬起而等到屏蔽结束后一起处理。本操作和关闭虚拟总中断的区别是，本操作完全是虚拟机内部的操作，无需唤起虚拟机监视器，因此在运行上高效得多。它一般被用于 RTOS 锁调度器的场合，通常而言用户无需主动调用。本操作没有嵌套计数，因此需要用户自行进行嵌套计数。

表 3-19 屏蔽虚拟中断向量处理

函数原型	void RVM_Virt_Mask_Int(void)
返回值	无。该操作总是成功。
参数	无。

3.6.4 解除虚拟中断向量处理屏蔽

本操作解除虚拟机内部对中断向量处理的屏蔽。虚拟机将立即处理任何的悬起中断。本操作没有嵌套计数，因此需要用户自行进行嵌套计数。

表 3-20 解除虚拟中断向量处理屏蔽

函数原型	void RVM_Virt_Unmask_Int(void)
返回值	无。该操作总是成功。
参数	无。

3.6.5 注册普通虚拟中断向量处理函数

本操作对一个普通虚拟中断向量注册处理函数。该处理函数将在该虚拟中断向量激活时被调用。在虚拟机启动时，所有的中断向量默认都未注册。

表 3-21 注册普通虚拟中断向量处理函数

函数原型	ret_t RVM_Virt_Reg_Vect(ptr_t Vect_Num, void* Vect)
返回值	ret_t 如果成功，返回 0。如果失败则会返回如下负值： RVM_ERR_RANGE 传入的虚拟中断源的中断向量号越界。
参数	ptr_t Vect_Num 虚拟中断源的中断向量号。 void* Vect 指向中断向量处理函数的函数指针。该函数推荐为 void (*)(void)类型；如果为其他类

	型，那么其参数的值是未定义的，返回值则会被直接抛弃。如果对该项传入 0[1]，那么意味着解除对该中断向量的注册。

3.6.6 注册时钟虚拟中断向量处理函数

本操作对时钟虚拟中断向量注册其处理函数。每当时钟中断发生时，该处理函数即被调用。通常而言客户机 RTOS 在初始化序列中会完成本操作而无需用户主动调用。

表 3-22 注册时钟虚拟中断向量处理函数

函数原型	void RVM_Virt_Reg_Timer(void* Timer)
返回值	无。该操作总是成功。
参数	void* Timer 指向时钟中断向量处理函数的函数指针。该函数推荐为 void (*)(void)类型；如果为其他类型，那么其参数的值是未定义的，返回值则会被直接抛弃。如果对该项传入 0[2]，那么意味着解除对时钟中断向量的注册。

3.6.7 注册线程切换虚拟中断向量处理函数

本操作对线程切换虚拟中断向量注册其处理函数。每当线程切换中断发生时，该处理函数即被调用。通常而言客户机 RTOS 在初始化序列中会完成本操作而无需用户主动调用。

表 3-23 注册线程切换虚拟中断向量处理函数

函数原型	void RVM_Virt_Reg_Ctxsw(void* Ctxsw)
返回值	无。该操作总是成功。
参数	void* Ctxsw 指向线程切换中断向量处理函数的函数指针。该函数推荐为 void (*)(void)类型；如果为其他类型，那么其参数的值是未定义的，返回值则会被直接抛弃。如果对该项传入 0[3]，那么意味着解除对线程切换中断向量的注册。

3.6.8 触发虚拟线程切换中断向量

本操作直接触发虚拟机的线程切换中断向量。如果虚拟机的总中断被关闭或者中断向量处理被屏蔽，那么线程切换中断向量将会悬起，待开启总中断或解除屏蔽后立即执行。

表 3-24 触发虚拟线程切换中断向量

[1] 空指针，也即 NULL

[2] 空指针，也即 NULL

[3] 空指针，也即 NULL

函数原型	void RVM_Virt_Yield(void)
返回值	无。该操作总是成功。
参数	无。

3.7 本章参考文献

无

第 4 章　移植 RVM 到新架构

4.1　简介

将 RVM 移植到其他架构的过程称为 RVM 本身的移植。在移植之前，首先要确定 RME 已经被移植到该架构。RVM 的移植分为两个部分：第一个部分是 RVM 本身的移植，第二个部分是 RVM 客户机库的编写。

本章的预定读者是那些需要移植 RVM 到新架构的开发者。

4.2　虚拟机监视器的移植

RVM 本身的移植包括其类型定义、宏定义、数据结构和函数体。RVM 的架构相关部分代码的源文件全部都放在 Platform 文件夹的对应架构名称下。如 ARMv7-M 架构的文件夹名称为 Platform/A7M。其头文件在 Include/Platform/A7M，其他架构依此类推。

每个架构都包含一个或多个源文件和一个或多个头文件。用户态库包含架构相关头文件时，总是会包含 Include/rvm_platform.h，而这是一个包含了对应架构顶层头文件的头文件。在更改 RVM 的编译目标平台时，通过修改这个头文件来达成对不同目标的相应头文件的包含。比如，要针对 ARMv7-M 架构进行编译，那么该头文件就应该包含对应 ARMv7-M 的底层函数实现的全部头文件。

在移植之前，可以先浏览已有的移植，并寻找一个与目标架构的逻辑组织最相近的架构的移植。然后，可以将这个移植拷贝一份，并将其当做模板进行修改。

4.2.1　类型定义

对于每个架构/编译器，首先需要移植的部分就是 RVM 的类型定义。在定义类型定义时，需要逐一确认编译器的各个原始整数类型对应的处理器字长数。RVM 的类型定义一共有如下五个：

表 4-1　RVM 的类型定义

类型	作用
tid_t	线程号的类型。这个类型应该被 typedef 为与处理器字长相等的有符号整数。 例子：typedef tid_t long;
ptr_t	指针整数的类型。这个类型应该被 typedef 为与处理器字长相等的无符号整数。 例子：typedef ptr_t unsigned long;
cnt_t	计数变量的类型。这个类型应该被 typedef 为与处理器字长相等的有符号整数。 例子：typedef cnt_t long;
cid_t	权能号的类型。这个类型应该被 typedef 为与处理器字长相等的有符号整数。 例子：typedef cid_t long;
ret_t	函数返回值的类型。这个类型应该被 typedef 为与处理器字长相等的有符号整数。 例子：typedef ret_t long;

为了使得底层函数的编写更加方便，推荐使用如下的几个 typedef 来定义经常使用到的确定位数的整形。在定义这些整形时，也需要确定编译器的 char、short、int、long 等究竟是多少个机器字的长度。有些编译器不提供六十四位整数，那么这个类型可以略去。

表 4-2 常用的其他类型定义

类型	意义
s8_t	一个有符号八位整形。 例如：typedef char s8_t;
s16_t	一个有符号十六位整形。 例如：typedef short s16_t;
s32_t	一个有符号三十二位整形。 例如：typedef int s32_t;
s64_t	一个有符号六十四位整形。 例如：typedef long s64_t;
u8_t	一个无符号八位整形。 例如：typedef unsigned char u8_t;
u16_t	一个无符号十六位整形。 例如：typedef unsigned short u16_t;
u32_t	一个无符号三十二位整形。 例如：typedef unsigned int u32_t;
u64_t	一个有符号六十四位整形。 例如：typedef unsigned long u64_t;

4.2.2 和 RME 配置项有关的、GSC 自动填充的宏

下面列出的宏和 RME 的内核配置有关。它们需要与 RME 在编译时的选项维持一致。关于这些宏的具体意义在此不加介绍，请参看 RME 的手册中的对应宏。GSC 在编译工程时会自动维持这些宏的一致性，因此在头文件中只要列出这些宏并为各个选项填充一个典型值就可以了。

表 4-3 和 RME 配置项有关的、GSC 自动填充的宏

宏名称	对应的 RME 宏
RVM_WORD_ORDER	RME_WORD_ORDER
RVM_KMEM_VA_START	RME_KMEM_VA_START
RVM_KMEM_SIZE	RME_KMEM_SIZE
RVM_KMEM_SLOT_ORDER	RME_KMEM_SLOT_ORDER

宏名称	对应的 RME 宏
RVM_MAX_PREEMPT_PRIO	RME_MAX_PREEMPT_PRIO

此外，还有一些和架构有关的配置宏。这些配置宏也需要和内核中编译时确定的相关宏维持一致，GSC 也会自动填充它们。

4.2.3 和 RME 配置项有关的、不变的的宏

下面列出的宏和 RME 的内核配置有关。它们需要与 RME 在编译时的选项维持一致。关于这些宏的具体意义在此不加介绍，请参看 RME 的手册中的对应宏。

表 4-4 和 RME 配置项有关的、不变的宏

宏名称	对应的 RME 宏
EXTERN	EXTERN
RVM_THD_WORD_SIZE	该项应填入线程内核对象的固定大小，单位为字长
RVM_PGTBL_WORD_SIZE_NOM(X)	RME_PGTBL_WORD_SIZE_NOM(X)
RVM_PGTBL_WORD_SIZE_TOP(X)	RME_PGTBL_WORD_SIZE_TOP(X)

此外，还有一些和架构有关的配置宏。这些配置宏也需要和内核中编译时确定的相关宏维持一致。

4.2.4 和 RVM 配置有关的、GSC 自动填充的宏

下面列出的宏和 RVM 的配置有关，可以根据 RVM 的配置状况灵活选择。它们会被 GSC 自动填充，因此在头文件中只要列出这些宏并为各个选项填充一个典型值就可以了。

表 4-5 和 RVM 配置有关的宏

宏名称	作用
RVM_MAX_PREEMPT_VPRIO	RVM 的虚拟机优先级数。必须是处理器字长的整数倍。 例子： #define RVM_MAX_PREEMPT_VPRIO 32
RVM_EVT_NUM	虚拟机监视器的事件源数量。必须是处理器字长的整数倍。 例子： #define RVM_EVT_NUM 20
RVM_MAP_NUM	虚拟机监视器允许的从事件源或物理中断源到虚拟机的最大映射数量。这个值必须不比事件源的数量小。 例子：

宏名称	作用
	#define RVM_MAP_NUM 32

此外，还有一些和架构有关的配置宏。GSC 也会自动填充它们。

4.2.5 和 RVM 配置有关的、不变的宏

下面列出的宏和 RVM 的配置有关，可以根据 RVM 的配置状况灵活选择。GSC 在编译工程时不会触碰这些宏及其值，因此移植者要负责填充它们。

表 4-6 和 RVM 配置有关的宏

宏名称	作用
RVM_VECT_NUM	该架构的中断源的数量。必须是处理器字长的整数倍。 例子： #define RVM_VECT_NUM 256

4.2.6 数据结构定义

需要移植的 RVM 的数据结构只有 RVM_Regs 一个。它是处理器的寄存器组结构体，用于在 RVM 中读取和修改虚拟机用户线程的寄存器组。这个结构体应当和 RME 中的完全一致。

4.2.7 汇编底层函数的移植

RVM 一共要实现以下 6 个汇编函数。这 6 个汇编函数主要负责 RVM 本身的启动和系统调用等。这些函数的列表如下：

表 4-7 RVM 移植涉及的汇编函数

函数	意义
_RVM_Entry	RVM 的入口函数。
_RVM_Jmp_Stub	线程创建和线程迁移用跳板。
_RVM_MSB_Get	得到一个字的最高位位置。
RVM_Inv_Act	进行线程迁移调用。
RVM_Inv_Ret	进行线程迁移返回。
RVM_Svc	调用 RME 的系统调用。
RVM_Thd_Sched_Rcv	接收线程的调度器事件。

这些函数的具体实现细节和注意事项会被接下来的各段一一介绍。

4.2.7.1 _RVM_Entry 的实现

本函数是 RVM 的入口，负责初始化 RVM 的 C 运行时库并跳转到 RVM 的 main 函数。本函数必须被链接到 RVM 映像的头部，而且其链接地址必须与 RME 内核编译时确定的 Init 线程的入口相同。

表 4-8 _RVM_Entry 的实现

函数原型	void _RVM_Entry(void)
意义	RVM 的入口。
返回值	无。
参数	无。

4.2.7.2 _RVM_Jmp_Stub 的实现

本函数用于在线程第一次运行或者线程迁移调用运行时，跳转到真正的入口。对于通常的架构，这个函数不是必需的，只要将它实现为直接返回即可，对于这些架构实际上我们也不会使用该函数；对于那些如 ARMv7-M 等中断退出时 PC 保存在用户栈上的架构，我们无法在创建新线程时通过设置线程的 PC 来确立返回地址。我们只能在内存中模拟一个返回堆栈，并借由返回堆栈中的 PC 值来设定将返回的地址。由于这个返回用堆栈是可能被反复使用的[1]，我们不希望频繁修改它内部所含的 PC 值。因此，我们将返回堆栈中的 PC 值设置为这个代码段。在这个代码段中，我们会给 PC 赋予线程创建时放在其他寄存器中的值，正确地向线程或者迁移调用传递其参数，并正式跳转到线程的入口执行。

表 4-9 _RVM_Jmp_Stub 的实现

函数原型	void _RVM_Jmp_Stub(ptr_t Entry, ptr_t Param)
意义	辅助跳转到正确的线程或迁移调用入口。
返回值	无。
参数	ptr_t Entry 线程的入口。
	ptr_t Param 线程的参数。

4.2.7.3 _RVM_MSB_Get 的实现

该函数返回该字最高位的位置。最高位的定义是第一个“1”出现的位置，位置是从 LSB 开始计算的[2]。比如该数为 32 位的 0x12345678，那么第一个“1”出现在第 28 位，这个函数就会返回 28。

[1] 比如我们创建 A 线程，然后销毁 A 线程，再试图在同一个栈上创建 B 线程

[2] LSB 算作第 0 位

表 4-10 _RVM_MSB_Get 的实现

函数原型	ptr_t _RVM_MSB_Get(ptr_t Val)
意义	得到一个与处理器字长相等的无符号数的最高位位置，也即其二进制表示从左向右数第一个数字“1”的位置。
返回值	ptr_t 返回第一个“1”的位置。
参数	ptr_t Val 要计算最高位位置的数字。

由于该函数需要被高效实现，因此其实现方法在不同的处理器上差别很大。对于那些提供了最高位计算指令的架构，直接以汇编形式实现本函数，使用该指令即可。对于那些提供了前导零计算指令（CLZ）的架构[1]，也可以用汇编函数先计算出前导零的数量，然后用处理器的字长-1（单位为 Bit）减去这个值。比如 0x12345678 的前导零一共有 3 个，用 31 减去 3 即得到 28。

对于那些没有实现特定指令的架构，推荐使用折半查找的方法。先判断一个字的高半字是否为 0，如果不为 0，再在这高半字中折半查找，如果为 0，那么在低半字中折半查找，直到确定第一个“1”的位置为止。在折半到 16 位或者 8 位时，可以使用一个查找表直接对应到第一个“1”在这 16 或 8 位中的相对位置，从而不需要再进行折半，然后综合各次折半的结果计算第一个“1”的位置即可。

4.2.7.4 RVM_Inv_Act 的实现

该函数进行一个标准线程迁移调用。在实现时，应当注意将参数按照内核调用约定按顺序放入正确的寄存器。

表 4-11 RVM_Inv_Act 的实现

函数原型	ret_t RVM_Inv_Act(cid_t Cap_Inv, ptr_t Param, ptr_t* Retval)
意义	进行一个标准的线程迁移调用。
返回值	ret_t 线程迁移系统调用本身的返回值。
参数	cid_t Cap_Inv 线程迁移调用权能的权能号。
	ptr_t Param 线程迁移调用的参数。
	ptr_t* Retval

[1] 如 ARM 等

	该参数用于输出，是一个指向存放线程迁移调用返回值的指针。如果传入 0（NULL），那么表示不接收由 RVM_Inv_Ret 返回的返回值。

由于 RME 的内核在线程迁移调用中不负责保存除了控制流相关寄存器之外的任何寄存器，因此用户态要将它们压栈。如果用户态没有用到其中的几个寄存器，那么可以不将这些没用到的寄存器压栈。这个函数应当被实现为一个标准的迁移调用，也即压栈所有的通用寄存器组，但是不压栈协处理器寄存器组。该函数还应当判断传入的 Retval 指针是否为 0（NULL），如果为 0 那么不输出线程迁移调用的返回值给 Retval 指向的地址。

4.2.7.5 RVM_Inv_Ret 的实现

该函数用于从线程迁移调用中返回。在实现时，应当注意将参数按照内核调用约定按顺序放入正确的寄存器。

表 4-12 RVM_Inv_Ret 的实现

函数原型	ret_t RVM_Inv_Ret(ptr_t Retval)
意义	从线程迁移调用中返回。
返回值	ret_t 如果成功，函数不会返回；如果失败，返回线程迁移返回系统调用的返回值。
参数	ptr_t Retval 线层迁移调用的返回值。如果调用端的 ptr_t* Retval 不为 0（NULL）的话，那么该参数将会被赋予调用端的*Retval。

4.2.7.6 RVM_Svc 的实现

该函数用于进行 RME 的系统调用。在实现时，应当注意将参数按照内核调用约定按顺序放入正确的寄存器。

表 4-13 RVM_Svc 的实现

函数原型	ret_t RVM_Svc(ptr_t Op_Capid, ptr_t Arg1, ptr_t Arg2, ptr_t Arg3)
意义	进行 RME 系统调用。
返回值	ret_t RME 系统调用的返回值。
参数	ptr_t Op_Capid 由系统调用号 N 和权能表权能号 C 合成的参数 P0。
	ptr_t Arg1

	系统调用的第一个参数 P1。
	ptr_t Arg2 系统调用的第二个参数 P2。
	ptr_t Arg3 系统调用的第三个参数 P3。

4.2.7.7 RVM_Thd_Sched_Rcv 的实现

该函数用于接收线程的调度器事件。由于本函数需要传出一个值，因此要采用和 RVM_Svc 不同的写法。其输出参数 Fault 输出的是线程迁移调用返回值寄存器中的值。

表 4-14 RVM_Thd_Sched_Rcv 的实现

函数原型	ret_t RVM_Thd_Sched_Rcv(cid_t Cap_Thd, ptr_t* Fault)
意义	进行 RME 系统调用。
返回值	ret_t 该系统调用的返回值。
参数	cid_t Cap_Thd 调度器线程的权能号。可以是一级权能号或者二级权能号。
	ptr_t* Fault 该参数用于输出，输出架构定义的线程错误原因。如果传入 0（NULL），那么表示不接收这个错误原因。

4.2.8 C 语言函数的移植

RVM 一共有五个 C 语言函数，涵盖了底层调试打印、线程栈初始化、进入低功耗状态和页表设置等等。这些函数的列表如下：

表 4-15 RVM 移植涉及的 C 语言函数

函数	意义
RVM_Putchar	底层打印函数，打印一个字符到控制台。
RVM_Stack_Init	初始化线程的线程栈。
RVM_Idle	将系统置于休眠状态。
RVM_Thd_Print_Fault	根据线程错误原因打印字符串到控制台。
RVM_Thd_Print_Regs	打印线程的寄存器组。

4.2.8.1 RVM_Putchar 的实现

本操作打印一个字符到控制台。在该函数的实现中，只需要重定向其输出到某外设即可。

表 4-16 RVM_Putchar 的实现

函数原型	ptr_t RVM_Putchar(char Char)
意义	输出一个字符到控制台。
返回值	ptr_t 总是返回 0。
参数	char Char 要输出到系统控制台的字符。

4.2.8.2 RVM_Stack_Init 的实现

本操作初始化线程的线程栈。部分架构如 ARMv7-M 要求在线程运行前将其栈初始化。这个函数用来初始化栈的内容。如果架构不要求线程运行前对栈进行初始化，那么这个函数可以实现为直接返回合适的栈地址。

表 4-17 RVM_Stack_Init 的实现

函数原型	ptr_t RVM_Stack_Init(ptr_t Stack_Base, ptr_t Stack_Size, ptr_t Entry_Addr, ptr_t Stub_Addr)
意义	初始化线程的线程栈。
返回值	ptr_t 用来初始化线程的寄存器组的（也即要向 RVM_Thd_Exec_Set 传入的）栈指针地址。
参数	ptr_t Stack_Base 栈区的起始地址。
	ptr_t Stack_Base 栈区的大小。
	ptr_t Entry_Addr 线程的入口。该参数未必会被使用。
	ptr_t Stub_Addr _RVM_Jmp_Stub 的入口位置。该参数未必会被使用。

4.2.8.3 RVM_Idle 的实现

本函数会由 RVM 的 Init 进程反复调用，在系统无负载时进入休眠状态。一般而言，该函数会直接调用一个 RME 内核功能，将处理器置为停止执行、等待中断的状态。

表 4-18 RVM_Idle 的实现

函数原型	void RVM_Idle(void)
意义	将系统进入休眠状态，直到下一个中断来临。
返回值	无。
参数	无。

4.2.8.4 RVM_Thd_Print_Fault 的实现

本操作根据 RVM_Thd_Sched_Rcv 返回的错误原因打印方便人类阅读的字符串到控制台。

表 4-19 RVM_Thd_Print_Fault 的实现

函数原型	void RVM_Thd_Print_Fault(ptr_t Fault)
意义	初始化线程的线程栈。
返回值	ptr_t 用来初始化线程的寄存器组的（也即要向 RVM_Thd_Exec_Set 传入的）栈指针地址。
参数	ptr_t Stack_Base 栈区的起始地址。
	ptr_t Stack_Base 栈区的大小。
	ptr_t Entry_Addr 线程的入口。该参数未必会被使用。
	ptr_t Stub_Addr _RVM_Jmp_Stub 的入口位置。该参数未必会被使用。

4.2.8.5 RVM_Thd_Print_Reg 的实现

本操作打印一个线程的所有寄存器组。寄存器的具体值需要使用 RME 扩展标准中定义的标准内核功能调用 RVM_KERN_DEBUG_REG_MOD 得到。

表 4-20 RVM_Thd_Print_Reg 的实现

函数原型	void RVM_Thd_Print_Fault(cid_t Cap_Thd)

意义	初始化线程的线程栈。
返回值	无。
参数	cid_t Cap_Thd 该线程的权能号。可以是一级编码或二级编码。
	ptr_t Stack_Base 栈区的大小。
	ptr_t Entry_Addr 线程的入口。该参数未必会被使用。
	ptr_t Stub_Addr _RVM_Jmp_Stub 的入口位置。该参数未必会被使用。

4.3 RVM 客户机库的移植

RVM 客户机库的移植需要修改三个文件，分别是 rvm_guest_xxx.c, rvm_guest_xxx.h 和 rvm_guest_xxx.s。在移植客户机时可以参考已经实现好的客户机库，并且将其拷贝一份作为新的移植的起点。

4.3.1 客户机库的组成部分和实现原理

客户机库主要负责客户机内部的 RVM 虚拟化扩展层服务。它可以从逻辑上主要分成三个部分：第一个部分是初始化部分，这一部分负责在虚拟机启动后初始化 RVM 的数据结构。第二个部分是虚拟化服务部分，这一部分负责向 RVM 发起超级调用完成特权操作。第三个部分是中断处理部分，这一部分负责处理 RVM 发向虚拟机的中断。

每个客户机都是单独的一个进程，并且拥有自己的权能表和页表。页表是在虚拟机创建之时就设置好的，因此无需客户机库关心；权能表在虚拟机被创建时也被创建好，而且内部有两个信号端点。

位于第一个位置[1]的信号端点是给虚拟机发起超级调用使用的。对于虚拟机而言，该端点只允许发送而不允许阻塞。一旦虚拟机向该端点发送了信号，那么阻塞在这个端点上的 RVM 调度器守护进程就会解除阻塞，抢占虚拟机的运行，并且从虚拟机和 RVM 的共享内存区读取参数开始执行超级调用。

位于第二个位置[2]的信号端点是给虚拟机的中断处理线程使用的。平时虚拟机的中断处理线程应该在该端点阻塞，其用户线程也可以向此端点发送。一旦有中断发生，该端点就会接受到从 RVM 的信号，此时中断处理线程要处理中断并负责维护用户线程的寄存器组。

4.3.2 类型定义

类型定义的移植和 4.2.1 节是完全相同的。在此我们不再赘述。

[1] 权能号为 0

[2] 权能号为 1

4.3.3 宏定义

宏定义的移植和 4.2.3 节是完全相同的。在此我们不再赘述。

4.3.4 汇编底层函数的移植

RVM 一共要实现以下 6 个汇编函数。这 6 个汇编函数主要负责 RVM 本身的启动和系统调用等。这些函数的列表如下：

表 4-21 RVM 移植涉及的汇编函数

函数	意义
_RVM_MSB_Get	得到一个字的最高位位置。
RVM_Inv_Act	进行线程迁移调用。
RVM_Inv_Ret	进行线程迁移返回。
RVM_Svc	调用 RME 的系统调用。
RVM_Thd_Sched_Rcv	接收线程的调度器事件。
RVM_Fetch_And	原子与操作。

这些函数的具体实现细节和注意事项会被接下来的各段一一介绍。其中，_RVM_MSB_Get、RVM_Inv_Act、RVM_Inv_Ret、RVM_Svc 和 RVM_Thd_Sched_Rcv 与 4.2.7 节提到的同名函数的实现是完全相同的。我们在这里仅介绍该节未提到的函数 RVM_Fetch_And。

该函数完成一个基本的逻辑与（Fetch-And-aNd，FAN）原子操作。在某些架构上它有直接的指令支持，此时可以考虑以汇编或内联汇编实现该指令。在某些 RISC 架构上，也可以考虑使用排他性加载和排他性写回指令来支持[1]。具体的支持方法随各个处理器而有不同。

表 4-22 RVM_Fetch_And 的实现

宏定义	ptr_t RVM_Fetch_And(ptr_t* Ptr, ptr_t Operand)
意义	进行逻辑与原子操作。该操作会把*Ptr 的值和 Operand 进行逻辑与，然后写回*Ptr，并且返回和 Operand 进行逻辑与之前的*Ptr。
返回值	ptr_t 与上 Operand 之前的*Ptr。
参数	ptr_t* Ptr 指向目标操作地址的指针。

[1] 如 ARMv7 的 LDREX 和 STREX 指令

ptr_t Operand

目标操作地址要与上的无符号数。

4.4 本章参考文献

无

第 5 章 附录

5.1 RVM 的已知问题和制约

RVM 存在一些已知的限制和制约。这些限制和制约将在下面一一说明。

5.1.1 无多核支持

RVM 暂时不支持多核微控制器。这是因为微控制器和微处理器的产品逻辑完全不同。微控制器多核的最大意义在于实时性和可靠性：不同的核心的功能在系统设计时静态分配，一旦一个核发生任何软硬件问题，哪怕甚至是硬件 BUG 导致的永久死锁或核心损坏，其他核心可以不受影响继续运行。为此，多核微控制器在的各核心差异往往很大，某些 CPU 支持 FPU，另一些则不支持；某些 CPU 支持 MPU，另一些则不支持。将 RME 和 RVM 移植到这些平台上是有可能的，但其易用性会受到很多损失。如果你对这样的应用有兴趣，请联系我们。

5.2 本章参考文献

无