

突越·中生·菊石（七阶）

Mutatus·Mesozoa·Ammonite(R.VII)

M7M2(Ammonite) R2T1

微控制器实时准虚拟机监视器

技术手册

系统特性

1.实时响应

- 在微控制器上提供优秀的虚拟化实时性能
- 虚拟机间实时性互不影响
- 同等功能支持下实时性最好的实现方案

2.全面的体系架构支持

- 可简易地在多种微控制器之间移植，底层汇编数量最小化
- 支持多种微控制器
- 支持多个量级的内存大小

3.高安全性

- 虚拟机沙盒之间物理隔离，提供安全保障
- 虚拟文件系统支持
- 轻量级网络协议栈支持

4.高灵活性

- 可以分别更新各个虚拟机
- 可以运行时动态更新虚拟机
- 可以进行虚拟机快照和恢复

目录

系统特性.....	1
目录.....	2
第一章 概述.....	3
1.1 简介.....	3
1.1.1 设计目的和指标.....	3
1.1.2 软件版权与许可证.....	4
1.1.3 主要参考系统.....	4
1.2 前言.....	4
1.2.1 原生型虚拟机监视器.....	5
1.2.2 宿主型虚拟机监视器.....	5
1.2.3 容器型虚拟机监视器.....	5
1.2.4 全虚拟化型虚拟机监视器.....	5
1.2.5 准虚拟化型虚拟机监视器.....	6
1.3 RVM 的组成部分.....	6
1.3.1 用户态库部分.....	6
1.3.2 中断服务线程部分.....	6
1.3.3 虚拟机监视器部分.....	6
第二章 用户态库部分.....	7
2.1 简介.....	7
2.2 权能表相关系统调用.....	7
2.3 内核功能相关系统调用.....	7
2.4 页表相关系统调用.....	7
2.5 进程相关系统调用.....	8
2.6 线程相关系统调用.....	8
2.7 信号端点相关系统调用.....	8
2.8 线程迁移相关系统调用.....	9
第三章 中断服务线程.....	10
第四章 移植 RTOS 到 RVM.....	11
第五章 移植 RVM 到新架构.....	12

第一章 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（电机控制）、非实时不可靠部分（网络协议栈、高级语言虚拟机）和某些对信息安全有要求的应用（加解密算法）等一起运行，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（RTOS）的内核和应用程序通常编译在一起，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，如 FreeRTOS/MPU，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（如内核定时器等）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 就是针对上述问题而提出的、面向高性能 MCU 的系统级虚拟化环境。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。它能够满足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安全运行而不发生信息安全问题。

这样，系统中安全的部分和不安全的部分就可以用不同的标准分开开发和认证，方便了调试，也节省了认证经费和开发成本，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软件包如 GUI 和语言虚拟机（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模组直接运行在虚拟机平台上，无需重新认证，这进一步节省了认证和开发成本。

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 设计目的和指标

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：

1.1.1.1 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（通常称为“固件”）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 高效的资源利用

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 资源静态分配，代码静态链接

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 不具备内存管理单元

与微处理器不同，微控制器通常都不具备内存管理单元（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（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具有很局限性的。

1.1.1.6 处理器单核居多

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控制器采取多核设计或者甚至不对称多核设计，在这些处理器上使用微控制器用户态库时，可以考虑将这两个核分开来配置，运行两套独立的操作系统，并且将处理器之间的中断作为普通的设备中断处理，也即两个处理器互相把对方看做自己的外设。

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 LGPL v3，但是对一些特殊情况（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）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：

RMP (@EDI)

RT-Thread (@睿赛德)

FreeRTOS (@Real-Time Engineering LTD)

虚拟机的设计参考了如下系统的实现：

Contiki (@Contiki Community)

XEN (@Cambridge University)

KVM (@Linux Community)

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%CPU 开销。

RVM 完全支持低功耗技术和 Tick-Less 技术，可以实现全系统无节拍工作，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。RVM 也支持可信计算基（TCB）技术，将系统在安全态和非安全态区分开，允许一些虚拟机运行在普通状态，另一些虚拟机运行在可信计算基状态。这样可以允许轻量级区块链（IOTA 等技术）等高敏感操作系统和应用运行在安全环境。

对于整个 RTOS 的虚拟化，可以使用 RVM 的宿主型虚拟机（虚拟机分类见下）技术，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。对于那些对实时性有极高要求的应用，可以使用 RME 本身支持的容器型虚拟机，更可以将处理代码写入中断向量，获取甚至快于小型 RTOS 的响应时间。如有多核处理器还可支持网络功能虚拟化（NFV）和定制硬件虚拟化，将网络功能或者定制 I/O 功能交给单独的一颗处理器来运行定制的虚拟机，从而大幅度增加网络吞吐。

RME 和 RVM 在全配置下功能下共占用最小 64kB 内存（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），外加固定占用 128kB 程序存储器，适合高性能微控制器使用。此外，如果多个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（FV）和准虚拟化型（PV）。

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（也即第一个启动的、具有管理特权的虚拟机）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（不包括其 KVM 版本）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

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（Intel VT-x、AMD AMD-V）处理，对于 I/O 也需要特殊硬件功能（Intel VT-d）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

RVM 由用户态库、中断服务线程和虚拟机监视器三个部分组成。关于这三个部分的介绍如下：

1.3.1 用户态库部分

这一部分是 RME 提供的系统调用的简单封装，它将原来的汇编格式 RME 系统调用接口封装成了 C 语言可以直接调用的形式。这一层被中断服务线程和虚拟机监视器同时依赖，如果需要编写 RME 的原生应用，那么也需要依赖于它。关于该部分的具体介绍以及各个参数的描述请参看 RME 的手册，在这里不加以额外介绍。

1.3.2 中断服务线程部分

这一部分是 RME 的中断服务线程。它们主要用来接受从中断向量来的信号，并对它们加以初步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。这一部分线程在 RME 中拥有最高的优先级。这一部分组件是高度安全的，并且和虚拟机完全隔离开来。即使系统中的其他虚拟机都已崩溃，这些组件仍能保证正常运行，从而维护被控设备的安全。

1.3.3 虚拟机监视器部分

这一部分是 RME 系统中优先级最低的部分。这一部分管理各个虚拟机，包括了错误处理线程、时间处理线程、中断处理线程和超调用处理线程、虚拟机中断向量线程和虚拟机用户线程六个线程。该部分是 RVM 的核心，实现了 RVM 的绝大部分功能。

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.

第二章 用户态库部分

2.1 简介

由于 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。关于这些函数的返回值和参数的具体意义请参见 RME 本身的手册。具体的函数列表如下所述：

2.2 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：

名称	接口函数原型
创建权能表	<code>ret_t RVM_Captbl_Crt(cid_t Cap_Captbl_Crt, cid_t Cap_Kmem, cid_t Cap_Captbl, ptr_t Vaddr, ptr_t Entry_Num)</code>
删除权能表	<code>ret_t RVM_Captbl_Del(cid_t Cap_Captbl_Del, cid_t Cap_Del)</code>
传递普通权能	<code>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)</code> 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	<code>ret_t RVM_Captbl_Pgtbl(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)</code> 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	<code>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)</code> 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	<code>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)</code> 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 64Byte。
权能冻结	<code>ret_t RVM_Captbl_Frz(cid_t Cap_Captbl_Frz, cid_t Cap_Frz)</code>
权能移除	<code>ret_t RVM_Captbl_Rem(cid_t Cap_Captbl_Rem, cid_t Cap_Rem)</code>

2.3 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：

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

2.4 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：

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 Vaddr, ptr_t Start_Addr, ptr_t Top_Flag, ptr_t Size_Order, ptr_t Num_Order)
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)
析构页目录	ret_t RVM_Pgtbl_Des(cid_t Cap_Pgtbl, ptr_t Pos)

2.5 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：

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_Kmem, cid_t Cap_Proc, cid_t Cap_Captbl, cid_t Cap_Pgtbl, ptr_t Vaddr)
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)

2.6 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：

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 Vaddr)
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 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, ptr_t Prio)
更改优先级	ret_t RVM_Thd_Sched_Prio(cid_t Cap_Thd, ptr_t Prio)
接收调度事件	ret_t RVM_Thd_Sched_Rcv(cid_t Cap_Thd)
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)

2.7 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：

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_Kmem, cid_t Cap_Sig, ptr_t Vaddr)
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)

2.8 线程迁移相关系统调用

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：

名称	接口函数原型
创建线程迁移	ret_t RVM_Inv_Crt(cid_t Cap_Captbl, cid_t Cap_Kmem, cid_t Cap_Inv, cid_t Cap_Proc, ptr_t Vaddr)
删除线程迁移	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)
激活线程迁移	ret_t RVM_Inv_Act(cid_t Cap_Inv, ptr_t Param, ptr_t* Retval)
迁移调用返回	ret_t RVM_Inv_Ret(ptr_t Retval)

本章参考文献

无

第三章 中断服务线程

第四章 移植 RTOS 到 RVM

第五章 移植 RVM 到新架构