M7M02 微控制器

嵌入式虚拟机监视器

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

微控制器嵌入式虚拟机监视器（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

·提供带图形化界面的工程生成器

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，任何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 仅物理地址空间

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，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 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。

RME 和 RVM 在全功能配置下共占用最小 16kB 内存[3]，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

[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

[3] 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

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，也叫 I 型虚拟机监视器。多个虚拟机在这些物理机的实例上运行。这类虚拟机监视器的优势在于高性能、低开销，但缺点则是自身不具备虚拟机管理功能，需要 0 域[1]对其他虚拟机进行管理，且无法像常规操作系统那样运行轻量级的普通进程。此类虚拟机监视器有 KVM[2]、Hyper-V、VMware ESXi 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，也叫 II 型虚拟机监视器。多个虚拟机在这些物理机的实例上运行。这类虚拟机的优势在于系统独立、自身具备一定的管理功能，并允许将它们的虚拟机进程和其它普通的宿主机进程一起调度，因此得到了最广泛的应用，但效率通常认为比原生型要低一些。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，其虚拟机也作为进程运行在 RME 上，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，严格来讲并不属于虚拟机监视器，但由于它也具有一定的资源分区功能而在某些场合被称为 III 型虚拟机监视器。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，这接口与操作系统自身的系统调用一致，而其它接口均不被兼容，因而普适性很差。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，也可以不使用 RVM 而把其提供的用户态库用来开发原生的容器型虚拟机应用。事实上，如果不在 RVM 配置文件中声明任何虚拟机，RVM 的虚拟化功能会在编译时被从系统中自动去掉以适应资源极端的环境[4]。

---

[1] 也即第一个启动的、具有管理特权的虚拟机

[2] KVM 模糊了原生型和宿主型的边界，严格地讲不属于其中任何一类

[3] 不包括其 KVM 版本

[4] 如 Cortex-M0+等 RAM 在 20KiB 上下的环境

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[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 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，并假定读者对虚拟化的基本概念有一般了解。在本手册中，所有的 typedef 类型之前的“rvm_”前缀均被省去。

本手册的第一部分为第 2 章。在该章节中将概略描述 RVM 的架构，推荐详细阅读。

本手册的第二部分为第 3 章到第 5 章。在这些章节中将分别详细描述 RVM 的技术细节，推荐按需查阅。其中，第 3 章介绍原生进程的细节信息，第 4 章介绍虚拟机的细节信息，第 5 章介绍工程生成器的细节信息。

本手册的第三部分为第 6 章。在该章节中将从用户角度详述 RVM 及其工程生成器的使用方式，推荐详细阅读。

本手册的最后部分为第 7 章。在该章节中将详细描述如何移植 RVM 到新架构。如果用户不需要进行移植，无需阅读该章节。

1.4 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.

[1] 如 Intel VT-x、AMD AMD-V 等

[2] 如 Intel VT-d 等

第 2 章　虚拟化架构

2.1　架构总览

RVM 是专为资源受限的微控制器打造的虚拟机监视器，它能够将一颗微控制器分解成时空上互不影响的多颗虚拟微控制器。即便其中一颗虚拟微控制器发生了致命故障，其它虚拟微控制器仍然能保持正常运作，不会受到不利影响，从而有效阻止故障或安全漏洞在整个系统中的扩散。

与常见的 KVM、Virtual Box、VMWare 等产品不同，RVM 将微控制器的资源静态分配给各颗虚拟微控制器，其资源分配信息在系统编译时就能够确定。三个重要因素导致了这种设计：（1）绝大多数微控制器依赖于 Flash 存储器保存程序，缺乏加载动态运行时库的能力，（2）微控制器的资源非常少，需要极为高效地进行利用，因此很难承担动态数据结构带来的额外开销，以及（3）在关键系统中引入过多动态性可能导致系统行为难以预测。当然，在各个虚拟微控制器内部，还是可以使用动态内存分配、动态线程创建等功能的；即便这些动态行为导致虚拟微控制器故障，也不会引起其它虚拟微控制器的正常运作。

RVM 与传统虚拟化解决方案的另一个不同在于其客户机运行实时操作系统（Real-Time Operating System，RTOS）。这些操作系统有两个特点：（1）它们要求系统时延是确定性的，（2）它们要求虚拟化的开销要非常低。常规的虚拟化解决方案往往是基于操作[1]翻译的，其底层原语往往离客户机操作系统的接口语义有一定偏离，一个简单的客户机请求就有可能被翻译成大量的超级调用（Hypercall），引起上百倍的时延放大。为了解决这个问题，RVM（1）内部的一切代码均是按照实时系统的标准设计的，其执行时延确定，（2）内部操作原语和超级调用的语义上被设计为 RTOS 友好的，一个 RTOS 操作仅会被翻译成一个操作原语或超级调用。

RVM 与传统虚拟化解决方案的最后不同在于其往往要面临（极端）苛刻的存储器要求。这些要求是相当极端[2]的，但并非不可理解：任何被用于资源隔离的 CPU 和存储器开销在那些不重视产品质量的设计中都可以节省；偏偏微控制器的价格非常低、用量非常高，任何一点增加的成本都能极大地反映在产品最终的售价上；如果隔离开销过高以至于需要哪怕稍贵一点的微控制器，就得不偿失了。为此，RVM 被设计为极端节约存储器。它自身只需要最少 5kB RAM 和 20kB ROM 即可运行，因而可能是世界上最小的虚拟化环境。这使得它可以普遍用于那些 128kB ROM、64kB RAM 的主流微控制器系列，只要它们具备内存保护单元（Memory Protection Unit，MPU）。

为了应对上述挑战，RVM 要求在系统编译时静态确定大量的地址信息，并创建一系列具备特定参数的内核对象。相应的数据结构也是特化设计的，这使得手工填充它们十分困难。为此，RVM 还提供了工程生成器，可以根据对系统的描述生成工程所需的初始化代码、链接器脚本甚至是对应于相应构建环境的工程文件。

---

[1] 或二进制

[2] 某些场合几十个甚至几个字节都要争取

这样，RME 微内核、RVM 监视器、运行在其上的原生进程和各 RTOS 虚拟机，以及用以生成整个工程的工程生成器构成了 RME 的微控制器操作系统生态。虚拟化的总架构见下图所示：图中实线代表硬件隔离机制，虚线代表软件模组的边界。

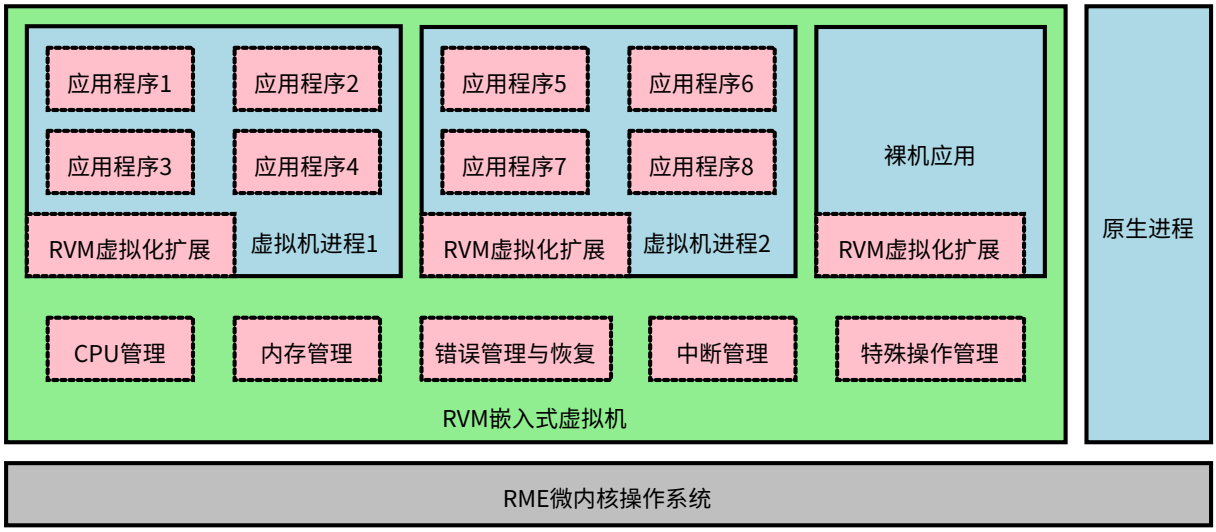

图 2-1 RVM 架构简图

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。

2.1.1 RME 微内核

RME 微内核作为 RVM 的底层支撑存在，负责向 RVM 提供基本的操作系统服务。其它一切运行在微控制器上的组件都是作为 RME 进程存在的。有关 RME 微内核，请参见相关内核文档。

2.1.2 RVM 虚拟机监视器

RVM 虚拟机监视器负责具体的虚拟机管理工作，它作为 II 型虚拟机监视器运行在 RME 上。具体地，它将物理中断和事件源信号下发到各个虚拟机，并响应来自各个虚拟机的超级调用请求。在虚拟机遇到异常时，RVM 还负责重启对应的虚拟机。虚拟机监视器包括五个守护线程：启动守护线程（Initialization Daemon，INIT）、安全守护协程（Safety Daemon，SFTD）、时间守护协程（Timer Daemon，TIMD）、向量守护协程（Vector Daemon，VCTD）和超级调用守护线程（Hypercall Daemon，HYPD），其中后三个协程，它们共同运行于虚拟化守护线程（Virtual Machine Monitor Daemon，VMMD）之上。下文将对这五个守护线程加以一一介绍。

2.1.2.1 启动守护线程 INIT

INIT 是 RME 在启动时创建的，由它负责初始化系统中的原生线程与虚拟机，并启动剩下的四个守护线程。在这一工作完成后，INIT 即进入休眠，并仅在系统中无可运行的线程时被调度。它一旦被调度，就反复执行处理器休眠命令，令处理器进入低功耗状态以等待中断来临。

2.1.2.2 安全守护协程 SFTD

SFTD 运行在系统的最高优先级[1]，负责监控原生进程和虚拟机的运行。它平时阻塞在错误处理端点 RVM_Sftd_Sig_Cap 上，一旦原生进程和虚拟机发生任何异常，SFTD 便解除阻塞开始运行。若发生异常的是原生进程，SFTD 将重启整个系统；若发生异常的是虚拟机，则 SFTD 仅重启该虚拟机。

2.1.2.3 向量守护协程 VCTD

VCTD 运行在较原生进程低，但较虚拟机高的优先级上，负责将系统中的物理中断和事件激活转送给各个虚拟机的中断源。它平时阻塞在中断向量端点 RVM_BOOT_INIT_VCT 上，一旦该端点收到任何信号，VCTD 便解除阻塞开始运行。VCTD 将从分别位于 RVM_PHYS_VCT_BASE 和 RVM_VIRT_EVT_BASE 的两个 RVM_Flag 数据结构中提取中断向量和事件源激活信息，并将其转送给虚拟机。

每个 RVM_Flag 数据结构都由两个相同的 struct __RME_RVM_Flag 合并而成。这是因为，VCTD 访问 struct __RME_RVM_Flag 时需要独占它，而此时内核若需要传递激活信息给 RVM，只能选择填充另一个结构体副本。每个结构体内部都有如下三个域：

表 2-1 struct __RME_RVM_Flag 中各个域的意义

域名称	作用
ptr_t Lock	决定此结构体是否正被 VCTD 使用。如果 VCTD 正在使用该结构体，内核无法对此结构体进行操作，否则可能引起竞争冒险。则内核会选择去填充另一个结构体。
ptr_t Fast	指示快速信号源的激活。如果为 1，则快速中断源激活。当前，这一个域仅在位于 RVM_PHYS_VCT_BASE 的结构体中使用，用来传达时钟中断信号。
ptr_t Group	指示被激活的慢速信号源的组别，是一个位图。若某位被置位，则意味着 Flag 域中该组别有被激活的信号：如果第 2 位被置位，则意味着 Flag[2] 的某一位不为 0。设置该域的目的是快速查找被激活的信号所在的组别。
ptr_t Flag[]	一个位图，指示具体的被激活的信号源。其长度由信号的数量决定：如果某处理器的字长为 32 位，而有 96 个中断源，则此域的长度为 96/32=3 个机器字长。

[1] Kern_Prio-1

比如，在 32 位处理器上，若 34 号物理中断源被激活，内核会首先查找位于 RVM_PHYS_VCT_BASE 的两个 struct __RME_RVM_Flag 结构体，并选择一个 Lock 域为 0 的副本。然后，内核会将 Flag[1]的第[2]个位置位，并将 Group 的第[1]个位置位。最后，内核会向 RVM_BOOT_INIT_VCT 发送信号，激活 VCTD。VCTD 被激活后，会交替查找两个 struct __RME_RVM_Flag 中存在的置位的位。它在查找时，首先将第一个 struct __RME_RVM_Flag 的 Lock 置 1，然后查询 Group 中哪些位被置位，并根据该信息查询对应的 Flag[]，得到具体的中断源编号。若激活的是 34 号中断源，则应当查询到 Group 的位[1]置位，并转去查询 Flag[1]；在 Flag[1]中则发现其位[2]置位。由此，VCTD 可以推算出被激活的中断源是第 34 号，并将登记了 34 号物理中断源的那些虚拟机激活。有关虚拟机的激活，请参见 2.2.4。

事件源的激活过程与物理中断源的激活过程是类似的。关于事件源，请参见 2.2.7。

2.1.2.4 时间守护协程 TIMD

时间守护线程 TIMD 运行在与 VCTD 相同的优先级上，负责将 RVM 时钟嘀嗒传递给各个虚拟机。它平时阻塞在时钟中断端点 RVM_BOOT_INIT_VCT 上，一旦该端点收到定时器中断就将该中断传递给所有虚拟机。TIMD 每在时钟中断端点 RVM_BOOT_INIT_VCT 上收到一次信号，就称作一个 RVM 时钟嘀嗒。每个虚拟机可以指定一个分频系数 Period[1]，在收到 Period 个 RVM 时钟嘀嗒后才会向虚拟机发送一个虚拟时钟中断。

时间守护线程还允许为每个虚拟机指定一个看门狗超时时间 Watchdog[2]。虚拟机可以使用超级调用 RVM_Hyp_Wdg_Clr 来投喂看门狗；如果该虚拟机在 Watchdog 个时钟嘀嗒后仍未喂狗，则判断该虚拟机陷入死循环，此时 TIMD 将重启它。

2.1.2.5 超级调用守护协程 HYPD

超级调用守护线程 HYPD 运行在与 VCTD 相同的优先级上，负责接受并处理各个虚拟机发来的超级调用。它平时阻塞在超级调用处理端点 RVM_BOOT_INIT_VCT 上，一旦该端点收到信号就处理当前虚拟机发起的超级调用。有关超级调用的具体处理方法，请参见 2.2.6。

2.1.3 保护域

RVM 提供两种保护域，分别是原生进程和虚拟机。两者的简要对比如下：

表 2-1 原生进程与虚拟机的对比

保护域	秘密性	完整性	可用性	可审计性	实时性	紧凑性	可靠性	易用性
原生进程	强	弱	弱	弱	强	强	弱	弱
虚拟机	强	强	强	强	弱	弱	强	强

[1] 请参见 4.2

[2] 请参见 4.2

2.1.3.1 原生进程

RME 原生进程直接运行在 RME 微内核上，它们不受 RVM 虚拟机监视器的制约，且资源消耗要比虚拟机低。它们主要用来接受从中断向量来的信号，并在进行初步处理后将这些信号转发给虚拟机。原生进程内的线程的时间片都是无限的，且在整个系统中拥有最高的优先级。即使 RVM 和它负责管理的所有虚拟机都已崩溃，原生进程内的线程仍能正常运行，从而维护被控设备的安全。

即便原生进程不是虚拟机，工程生成器仍对它们提供了完善的支持，RVM 的用户态库也提供一系列对系统调用的封装以方便在 C 程序中使用系统调用。工程生成器甚至可以生成不含虚拟机的工程，此时只能使用原生进程和系统调用来完成应用程序的功能。

RVM 没有为原生进程提供容错功能；如果任何一个原生进程发生异常，则 RVM 的 SFTD 将认为系统中出现了严重的问题，并重启整个系统。因此，原生进程仅具备安全三要素中的秘密性，一旦被入侵，则系统的完整性和可用性将荡然无存。

2.1.3.2 虚拟机

每个 RVM 虚拟机都是一个单独的进程，其内部可以模拟 RTOS 运行时依赖的硬件抽象层。它具备一个中断向量线程 Vct 和一个用户程序线程 Usr 来分别模拟处理器中断和正常的程序执行流：为方便 RTOS 的软中断线程切换，Usr 线程可以直接触发 Vct 线程的运行，且 Vct 线程可以直接修改 Usr 线程的寄存器组。

在基于 RVM 的系统中，推荐将任何稍复杂的功能，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中的 RTOS 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仅会自动重启自身而非整个系统。这对于功能安全中的故障隔离是至关重要的。

虚拟机可以通过超级调用来访问一系列与虚拟化相关的特殊功能，包括中断管理、事件源管理与通信等。这些超级调用将被 RVM 的 HYPD 拦截并响应。

2.1.4 工程生成器

RVM 提供的工程生成器能够根据工程描述文件生成可编译的工程。它能够自动计算并分配各个内核对象的地址，并生成对应的配置文件，无需用户做任何手动调整。

2.2 虚拟化原理

在 RVM 启动后，第一个执行的是启动守护线程 INIT。它会加载各个虚拟机。并负责启动安全守护线程、向量守护线程、超调用守护线程和时间守护线程。其中，安全守护线程会在虚拟机发生错误时重启它[1]，向量守护线程会把来自设备的物理中断或事件源的激活重定向给对应虚拟机的虚拟中断向量，超调用守护线程负责处理超级调用和虚拟机调度；时间守护线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。

[1] RVM 不原生支持用户自定义错误处理

2.2.1 优先级配置

在 RVM 中，安全守护线程 SFTD 永远运行在系统的最高优先级。紧随其后的是原生进程中的线程，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。再接下来是 VCTD、TIMD 和 HYPD 三个守护线程，它们运行在比原生进程低但比虚拟机高的优先级上。然后则是在运行的虚拟机的中断处理线程 Vct 和用户线程 Usr，其中 Vct 的优先级又要高于 Usr。再之后是系统本身的 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：

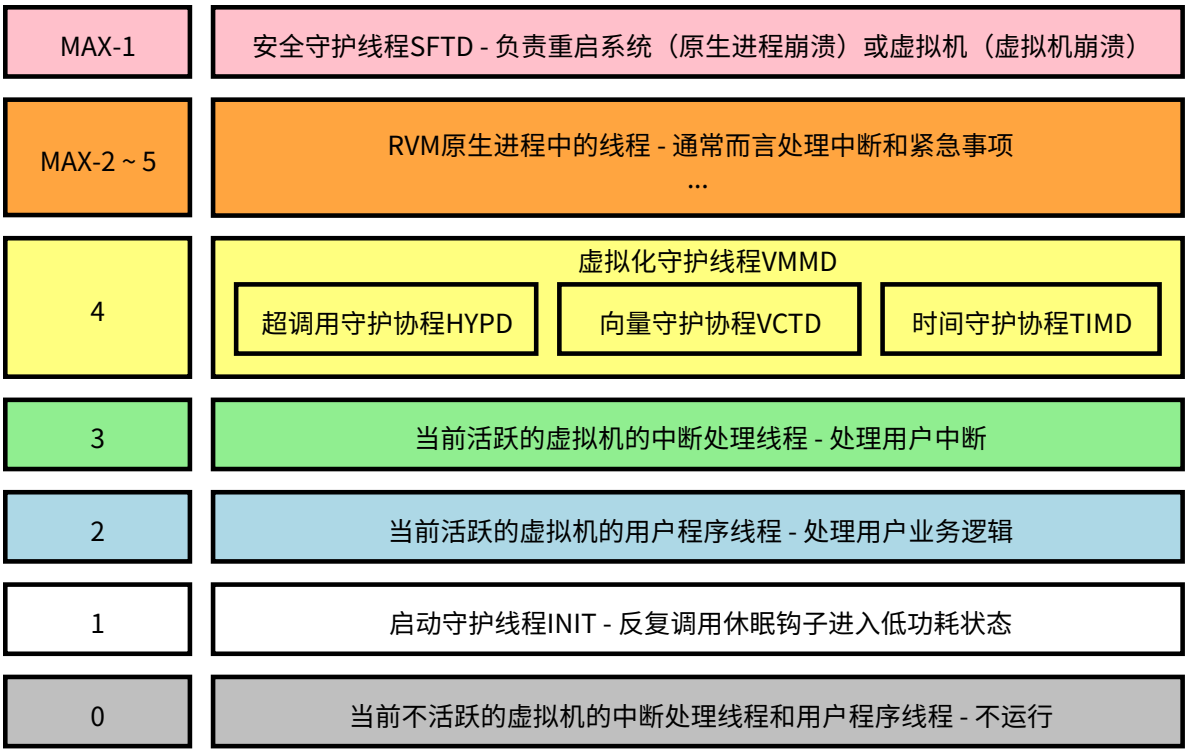

图 2-2 虚拟化体系的中断优先级

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线程决定的。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。

2.2.2 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，每个虚拟机的私有内存和共享内存，都是在编译时静态指定的。在系统开始运行之后，不再允许修改内存映射。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。

手动构造和填充页表是非常繁琐的；为了避免不必要的麻烦，工程生成器提供了页表自动构造和填充的相关功能。

2.2.3 CPU 的虚拟化

在不同的准虚拟化平台中，CPU 的虚拟化主要有两种主流设计策略，分别如下：

1. 将每个虚拟机作为一个线程运行。这是专用准虚拟化平台最常见的解决方案，Xen 即采取该策略。在每个客户机中，线程平时执行应用程序的代码；一旦设备中断或异常发生[1]，虚拟机监视器即强制修改线程的执行地址，迫使其执行一段中断处理代码，待处理结束后该线程自行返回应用程序中执行。在虚拟机监视器看来，此种方法最为方便，一方面每个客户机只有一个执行上下文，另一方面激活中断只要修改该上下文的指令指针。然而，在客户机看来，中断结束（End-Of-Interrupt，EOI）是否需要特意通知虚拟机监视器的问题会变得很难处理。这里主要有两种情况[1]：相对简单的（1）虚拟机监视器一旦迫使某客户机执行虚拟中断向量，在客户机使用专用的超级调用通知虚拟机监视器当前中断已结束之前，虚拟机监视器不得再修改指令指针以迫使虚拟机执行中断向量，和较为复杂的（2）虚拟机监视器会无视客户机是否正在执行虚拟中断向量而总是修改指令指针迫使虚拟机执行中断向量。在情况（1）中，客户机端需要调用超级调用来通知虚拟机，这无疑增加了一次超级调用。对于直接运行在特权态的原生型虚拟机监视器 Xen 而言，一次超级调用就是一次上下文切换，开销还是可以忍受的；对于运行在用户态的宿主型虚拟机监视器 RVM 而言，一次超级调用意味着两次上下文切换，引入的延迟就不可忽视了。如果采取情况（2）中的解决方案，则更加糟糕：这意味着中断向量入口是必须可重入的，因此需要一段非常特殊的汇编代码。这段代码必须负责在重入的执行实例执行到真正的中断向量之前识别重入且在此后主动退出，在退出的过程中还要涉及对栈框的修补以使执行流带着正确的栈指针返回正在执行的中断向量。一方面这段代码非常难以正确编写[2]，另一方面它的开头总是大片的压栈。如果发生一系列紧密连续的中断，开头的压栈代码就会在来得及进行栈框修补之前重入多次，在嵌入式环境下会导致潜在的栈溢出。因此，即便（2）在 Xen 中非常高效，我们也不采用这种方式。

2. 将每个客户机中的每个线程都作为一个独立的虚拟机线程运行。这是基于微内核的虚拟化的常见方案，L4Linux 即采用该策略[2]。在微控制器上，这种策略的问题是（1）需要将原客户机操作系统的线程接口想办法对接到微内核的系统调用，这需要充分理解原操作系统的设计，甚至可能涉及到线程控制块等架构不相关部分的修改，（2）将每一个客户机线程都对应到微内核线程会导致创建出大量的微内核线程对象，这在 Linux 等线程结构本就庞大的重量级操作系统中是没有问题的，但在线程普遍极为轻量的 RTOS 中就等于成倍放大了存储器消耗，（3）微内核的线程切换需要经过一系列操作权能检查、时间预算检查和保护域切换检查，而甚至不具备保护域的 RTOS 线程根本用不着这些检查，每次都调用微内核系统调用来进行线程切换相当于成倍放大了执行时间消耗，在那些把时间资源作为独立可分配资源的微内核[3]上就更是如此。

为了解决上面的困境，RVM 采用双线程的方法进行客户机的 CPU 虚拟化。每个客户机都具有 Vct 和 Usr 两个线程，Vct 线程负责中断处理，Usr 线程则负责处理用户程序。Vct 平时不运行，只有当接收

[1] 在 x86 上，Xen 利用了硬件特有的 Ring1 来加速系统调用和异常的处理，并不经过 Xen，但设备中断依然需要经过 Xen 进行分发

[2] 对五花八门的非标准嵌入式架构就更是如此

[3] 如 Composite 和某些 L4 变体等

到了来自 RVM 的中断触发信号时才运行。当中断处理完成后，Vct 又阻塞，Usr 开始运行。Vct 到 Usr 的切换利用了微内核的高效异步信号机制，不涉及 RVM，因此比超级调用快得多。

在 Usr 被切换走时，其寄存器组会被保存在由 Reg_Base[1]指定的地址。因此，要在 Vct 中修改 Usr 的寄存器组，只要修改那个地址上的寄存器数据即可。利用这个方法，可以在无需理解原 RTOS 设计的基础上对其硬件抽象层、尤其是其线程切换汇编进行逐行替换，从而轻松模拟 RTOS 中非常常见的线程上下文切换。

2.2.4 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中断处理线程 Vct，该线程的优先级比处理用户程序的 Usr 线程要高。该线程平时阻塞在各个虚拟机专属的虚拟中断向量端点上[2]，直到收到 RVM 发来的虚拟中断信号才被唤醒运行。

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；线程切换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。

每个虚拟机都维护一个 RVM_State 数据结构，其地址被 State_Base[3]指定。其内部的域如下：

表 2-2 struct RVM_State 中各个域的意义

域名称	作用
ptr_t Vct_Act	指示当前虚拟机中运行的线程。如果置 1，说明 Vct 线程正在运行，否则说明 Usr 线程正在运行。
struct RVM_Param Vct	Vct 线程专用的超级调用参数区。
struct RVM_Param Usr	Usr 线程专用的超级调用参数区。
struct RVM_Vctf Flag	指示当前虚拟机中被激活的虚拟中断向量。

在该数据结构中，与中断虚拟化相关的数据结构为 struct RVM_Vctf Flag，它的各个域如下表所示。关于其它域，请参见 2.2.6。

表 2-3 struct RVM_Vctf 中各个域的意义

域名称	作用
ptr_t Tim	指示时钟中断激活。

[1] 请参见 4.2

[2] 该端点在每个虚拟机内部都位于权能号 1 的位置

[3] 请参见 4.2

域名称	作用
ptr_t Ctx	指示线程切换中断激活。
u8_t Vct[]	指示其它虚拟中断向量激活。每个字节对应一个虚拟中断向量。

在该数据结构中，时钟中断和线程切换中断的激活由两个单独的机器字指示，其它虚拟中断向量则每个对应于一个字节。当对应的中断被激活时，相应的域就被置 1。这里不采取一个二进制位对应一个中断向量的方法，这样就不需要原子逻辑与指令[1]。由于每个虚拟机的虚拟中断向量数都有限，所以造成的存储空间浪费是可以容忍的。

比如，若 VCTD 判定某虚拟机的 4 号中断向量被激活，它会将 Vct[4]置 1，并向该虚拟机的 Vct 线程的专属端点发送信号。收到该信号的 Vct 线程被激活后，会先扫描整个结构体，并通过 Vct[4]被置 1 判断 4 号虚拟中断向量被激活。之后，Vct 线程会先将 Vct[4]清零，再调用用户使用 RVM_Virt_Vct_Reg[2] 注册的 4 号虚拟中断向量。这样一来，即便用户注册的中断向量执行时有新的 4 号中断到来，也不会产生中断丢失，因为 Vct[4]将重新置位：Vct 线程将在下一个循环后检测到 4 号中断悬起并再次进入它。

虚拟机的中断开关，以及虚拟机的虚拟中断向量与对应物理中断向量或事件源的对应关系是可以通过超级调用设置的。有关这些超级调用的信息请参见 4.4。一个标准的虚拟机中断初始化流程是：

（1）调用 RVM_Virt_Tim_Reg 注册时钟中断，

（2）调用 RVM_Virt_Ctx_Reg 注册线程切换中断，

（3）调用 RVM_Virt_Vct_Reg 注册其它虚拟中断，

（4）调用 RVM_Hyp_Vct_Phys 将所需的物理中断源映射到虚拟中断源，

（5）调用 RVM_Hyp_Vct_Evt 将所需的事件源映射到虚拟中断源，

（6）调用 RVM_Hyp_Vct_Lck 将事件源映射锁死以防止后续篡改，

（7）在合适的地方调用 RVM_Hyp_Int_Ena 开启虚拟机总中断。如果虚拟机还使用看门狗，需要调用 RVM_Hyp_Wdg_Clr 来启动看门狗。看门狗一经启动则无法关闭。

（8）虚拟机内的 RTOS 在其空闲线程中应当反复调用 RVM_Hyp_Vct_Wait 以使虚拟机在空闲时进入休眠，并等待下一次中断来临。如果虚拟机还使用看门狗，需要在合适的地方调用 RVM_Hyp_Wdg_Clr 来喂狗。

2.2.4.1 定时器中断的虚拟化

在 RVM 上运行的 RTOS 按照对定时器的运用方式可以分为两类，分别是有节拍（Periodic）内核和无节拍（Tickless）内核。前者要求系统中具备一个周期性发送中断的自动重装（Auto-reload）定时器，后者则要求每次上下文切换时按需编程一个倒计时结束后发送中断的一次性（One-shot）定时器。在裸

[1] VCTD 在置位某个向量时，Vct 线程可能正在复位其中的某个向量。如果采取每个二进制位对应一个中断向量，则 Vct 线程必须使用原子逻辑与操作来复位标志位。部分架构，如 ARMv6-M，是没有这种原子操作的，因此一个字节对应一个标志兼容性更好。

[2] 请参见 4.5.5

机上，后者在低功耗特性方面更加优越，因为在处理器休眠或仅调度一个任务时不产生时钟中断。为此，在那些支持无节拍内核的 RTOS 上，我们常常启用这个特性。

然而，在虚拟化环境中则正好相反。无节拍内核对定时器的编程不再能直接完成，而要通过超级调用求助 RVM。这会导致在每次上下文切换时多出一个超级调用，从而造成难以忍受的性能和实时性开销。此外，并非所有的 RTOS 都支持无节拍内核模式：哪怕有一个有节拍内核运行在 RVM 上，RVM 还是要周期性地被唤醒以发送定时器中断给它，从而不可能降低功耗；这让其它内核的无节拍特性变得徒增开销。退一步讲，即便运行在 RVM 上的一切虚拟机都是无节拍的，它们对定时器的编程次数的总和也会非常多，而 RVM 必须将硬件定时器的超时时间设置成所有时刻中的最小值。这会导致定时器总是在一小段时间后就被触发，也不太可能起到低功耗效果。

基于上述原因，RVM 使用有节拍设计。它需要的周期性时钟信号由 RME 的底层定时器产生，并且通过 RVM_BOOT_INIT_TIM 端点发送到 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，并且根据各个虚拟机配置的分频系数 Period[1]给各个虚拟机发送时钟中断。

具体地，Vct 线程将调用用户使用 RVM_Virt_Tim_Reg[2]注册的线程切换钩子 Tim。该钩子函数可以直接使用原裸机 RTOS 的定时器中断向量，且在需要时可使用 RVM_Virt_Yield[3]来触发上下文切换[4]。

2.2.4.2 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。这个中断的响应效率将直接决定虚拟机的线程切换时间，因此保证它的效率非常重要。为此，RVM 被特别设计为可以由用户线程 Usr 在不经过虚拟机监视器介入的情况下使用 RVM_Virt_Yield[5]直接触发 Vct 线程主导的上下文切换，且在 Vct 线程中可直接修改提前映射到 Reg_Base[6]的 Usr 寄存器组，无需再使用任何系统调用。这个实现最大限度地贴近了常见 RTOS 的线程切换语义，大大降低了虚拟机内部的线程切换开销。

具体地，Vct 线程将调用用户使用 RVM_Virt_Ctx_Reg[7]注册的线程切换钩子 Ctx。在该钩子中将实现对 Usr 的各个寄存器的保存和恢复，从而完成 RTOS 线程的切换。常见的线程切换手段包括两个步骤，（1）将一切寄存器压到老线程栈上，（2）再从新线程栈上恢复一切寄存器；这个过程在裸机 RTOS 中通常是由汇编语言完成的。在 RVM 中，实现该流程的 Ctx 钩子则可由 C 语言写成[8]。

---

[1] 请参见 4.2

[2] 请参见 4.5.7

[3] 请参见 4.5.7

[4] 就好像在原 RTOS 中触发 PendSV 等上下文切换中断那样

[5] 请参见 4.5.7

[6] 请参见 4.2

[7] 请参见 4.5.7

[8] 具体例子可以参见 RMP 的 RVM 移植

与裸机 RTOS 不同，寄存器的保存和恢复必须包括 Usr 线程使用的超级调用参数缓冲区，它位于 struct RVM_State 结构体的 struct RVM_Param Usr 域。这是因为，Usr 线程对应的多个 RTOS 线程可能会并发访问它[1]。关于该域的详细信息，请参见 2.2.6。

2.2.4.3 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。

2.2.5 设备的虚拟化

在不同的虚拟化平台中，设备及其驱动的虚拟化主要有四种主流设计策略，分别如下：

1. 将设备驱动放进一个单独的、具备特权虚拟机 Dom0 中，其余虚拟机通过向该虚拟机发送请求来访问设备。这是 Xen 等原生型虚拟机监视器常采取的策略[1]。采取这种策略的动力很好理解：（1）将设备直接交给虚拟机是无法安全共享的，要将它们与虚拟机隔离就需要将其放进单独的保护域，而虚拟机监视器提供了虚拟机作为隔离手段，（2）客户机操作系统中已有现成的驱动，单独为虚拟机监视器再准备一套驱动无异于重造轮子。然而，这种策略使任何设备 I/O 都包含昂贵的虚拟机间上下文切换，性能开销极大。此外，一旦虚拟机监视器对 Dom0 的中断调度或分发不当，其延迟将会扩散到整个系统中，造成毁灭性的实时性问题[3]。因此，RVM 不能采取这种策略。

2. 将设备驱动放在虚拟机监视器所在的内核态中。这是 VMware ESXi 等原生虚拟机监视器采取的另一种策略。这种方案可以完全规避掉 Dom0 的性能问题，但它将虚拟机监视器——也即整个系统的可信计算基——置于恶意驱动的威胁之下。显然，只有在虚拟机监视器厂商对驱动非常自信的情况下才能这样做，而且第三方设备厂商还必须配合这种认证。在微控制器中这些条件都不存在，因此 RVM 不能采取这种策略。

3. 将虚拟机监视器和操作系统内核做成一个，这样客户机自然就能享受操作系统自带的设备驱动。这是 KVM 等虚拟机监视器通常采取的策略。它和上一种策略实际上或多或少是同样的，只不过驱动程序的开发者现在普遍认为自己在给 Linux 操作系统开发驱动，因而接受度较高些。

4. 将各个驱动程序直接放置在各客户机中。这是很多工业场景和功能安全场景下的解决方案，因为工业场景下的每个虚拟机往往是各司其职[2]，各客户机间的功能重叠面很小，硬件的共享面也很小。利用工业嵌入式场景的这个特性，这个方案在设备划分得当时往往非常高效，因为虚拟机监视器将不再介入除中断分发之外的任何设备操作。然而，这种方案要求（1）任何设备的错误甚至是恶意操作都不得对系统安全和实时性造成威胁，包括那些往往被忽视的、由侧信道引起的问题[3]，（2）任何设备都只能被

[1] 比如它们同时进行超级调用

[2] 可以说是沿功能的分界面进行“纵切”，而非像大多数系统那样沿抽象层次的分界面进行“横切”

[3] 最典型的是 DMA 控制器，允许不可信虚拟机任意操作它可以轻易绕过所有安全机制

一个虚拟机独占，无法在虚拟机间进行任何设备共享，而且（3）中断仍然要传递到虚拟机中才可以处理，实时性不够。因此，这种方案不是总能使用。

考虑到微控制器的实际情况，RVM 鼓励在任何可能的时候使用方案 4。如果实时性不够，或者需要共享设备，可以考虑将驱动程序的简单、可信赖的上半部[1]放置在中断响应极快的原生进程中，而将其下半部和协议栈放置在客户机中。原生进程可以和客户机都共享一部分内存以用于传递数据，且在共享时要保证客户机不破坏原生进程中线程执行所必要的数据。

2.2.6 超级调用的设计

虚拟机可以通过超级调用向虚拟机监视器请求服务。有关这些超级调用的信息，请参见 4.4。要发起超级调用，虚拟机会先将超级调用所需的各个参数写到与 RVM 约定好的参数缓冲区，然后向 HYPD 的信号端点 RVM_Hypd_Sig_Cap 发送信号[2]。该参数缓冲区位于 RVM_State 数据结构中，且存在两份：struct RVM_Param Vct 为 Vct 线程专用，而 struct RVM_Param Usr 为 Usr 线程专用，这样就可以防止 Vct 线程和 Usr 线程在参数缓冲区上产生竞争冒险。

HYPD 在收到超级调用后，会通过 RVM_State 数据结构中的 Vct_Act 判断当前正在运行的线程。如果 Vct_Act 为 1，说明当前运行的是 Vct 线程，超级调用参数应当从 Vct 线程的参数缓冲区中提取；否则说明当前运行的是 Usr 线程，超级调用参数应当从 Usr 线程的参数缓冲区中提取。在执行完超级调用之后，HYPD 会将返回值写回对应的参数缓冲区；虚拟机在重新获得运行权后将提取该值作为超级调用的返回值。

由于一个 Usr 线程对应的多个 RTOS 线程同时也可能并发访问 Usr 线程的参数缓冲区，因此在 RTOS 的线程切换中，Usr 线程使用的参数缓冲区将和寄存器组一起压栈弹栈，这确保了不会发生竞争冒险。

2.2.7 跨保护域通信

为安全起见，原生进程与原生进程之间、原生进程与虚拟机之间，以及虚拟机与虚拟机之间是默认无法互相发送信号的。要使它们可以互相通信，需要做一些配置以在各个保护域中置入合适的信号原语。

2.2.7.1 从原生进程发送信号到原生进程

原生进程之间可以通过 RME 的异步信号机制来进行通信。一个进程中的线程在信号端点上阻塞后，另一个进程中的线程只要向该端点发送信号就可以将阻塞解除。这要求一个进程具备端点的接收权能，另一个进程具备端点的发送权能。相关信息请参见 3.5.6。

工程生成器可以根据用户对原生进程的描述正确地生成这种权能布局。只需要在接收的进程中声明接收端点，然后在发送的进程中向该端点发送就可以了。

[1] 通常只包括寄存器的简单读写以及消息队列的发送

[2] 该端点在每个虚拟机内部都位于权能号 0 的位置

2.2.7.2 从虚拟机发送信号到原生进程

若要从虚拟机发送给原生进程，则可直接在原生进程中声明接收信号端点，并在虚拟机中声明发送信号端点，由虚拟机直接向原生进程发起信号发送即可。相关信息请参见 3.5.6。

工程生成器允许在虚拟机中引用任何原生进程中声明的接收端点，并能够在系统启动时将对应的端点发送权限正确赋予虚拟机；虚拟机只需要向该端点发送就可以了。

2.2.7.3 原生进程或虚拟机发送信号到虚拟机

由于虚拟机的 Vct 和 Usr 线程是不允许阻塞在信号端点上的，虚拟机内也无法定义信号端点发送权限之外的任何其它内核对象，从其它保护域[1]到虚拟机的通信只能采取激活虚拟中断的方法。为此，RVM 引入了“事件源”的概念[2]：每个虚拟机都可以将自己的虚拟中断向量通过 RVM_Hyp_Vct_Evt 自由注册到事件源，该事件源一旦激活，注册了该中断源的虚拟机就都会收到该虚拟中断。

原生进程的优先级高于虚拟机，因此限制它们对事件源的发送没有意义[3]。虚拟机与虚拟机之间则并非如此：在虚拟机启动后，首先要通过 RVM_Hyp_Evt_Add 函数事先申请向事件源进行发送的权限，并在申请完成后使用 RVM_Hyp_Vct_Lck 将所有的事件源发送权限[4]锁死，在锁死后则不再允许对权限进行任何修改。此设计是基于这样一个假设：此后，通过 RVM_Hyp_Evt_Snd 即可向事件源进行发送。相关信息请参见 4.4。

若要从虚拟机发送给原生进程，则可直接在原生进程中声明接收信号端点，并在虚拟机中声明发送信号端点，由虚拟机直接向原生进程发起信号发送即可。有关这些函数，请参见 3.5.6。

事件源的总数量和事件源/物理中断源到虚拟中断的映射总数在工程生成器中可以配置，它是在工程编译前就静态决定好的；具体的映射关系则是在运行时动态建立的。

2.3 本章参考文献

[1] D. Chisnall, The definitive guide to the Xen hypervisor. in Prentice Hall open source software development series. Upper Saddle River, NJ Munich: Prentice Hall, 2008.

[2] A. Lackorzynski, “L4Linux Porting Optimizations” Master Thesis.

[3] S. Yoo, K.-H. Kwak, J.-H. Jo, and C. Yoo, “Toward under-millisecond I/O latency in Xen-ARM,” in Proceedings of the Second Asia-Pacific Workshop on Systems, Shanghai China: ACM, 2011.

[1] 无论该保护域是原生进程还是虚拟机

[2] 和常规的物理中断源一样，事件源也是一种全局刺激来源

[3] 它们只需要一直抢占处理器就可以阻止虚拟机监视器以及虚拟机的运行，因此限制它们没有意义

[4] 以及事件源/中断源到虚拟中断之间的映射关系

第 3 章 原生进程

3.1 简介

原生进程直接运行在 RME 微内核之上，因此其运行速度最快，开销最小。RVM 虚拟机监视器仅仅负责用 INIT 启动它们，而不能对它们提供任何服务。因此，原生进程仅能使用 RME 本身的系统调用，不能使用 RVM 超级调用。为了方便用户使用 RME 系统调用，RVM 提供的用户态库对它们做了 C 语言接口封装。这些接口都是线程安全的。

工程生成器可以生成包含原生进程的工程。一个工程甚至可以仅仅包含原生进程，此时需要用原生进程直接完成应用程序的功能。

3.2 进程描述符

每一个原生进程（和虚拟机）都有一个位于其首部的进程描述符，该头部记载了进程中的每一个可能的入口点。这些入口点被虚拟机监视器识别，以用来初始化其内部数据结构。进程头的组成如下表所示。

表 3-1 进程描述符的结构

地址（字）	内容
0	魔法数字，对于原生进程而言是 0x49535953，对于虚拟机而言是 0x56495254。
1	进程头的入口表项数 N。
2	第 0 个入口表项。
3	第 1 个入口表项。
…	…
N+1	第 N-2 个入口表项。
N+2	第 N-1 个表项，固定存放跳板。在常规架构上，该项无用；在那些从栈中恢复 PC 的架构[1]上，一段跳板是必须的，该跳板负责跳转程序到真正的入口点去执行。有关该跳板的描述请参见 6.2.6.2。

工程生成器可以生成此进程描述符，当然也可以手动对其进行填充。

3.3 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。当权限关系十分复杂时，这是十分繁琐、容易出错的。工程生成器可以生成高度优化的数据结构自动完成

[1] 如 ARMv7-M

上述流程，大大降低配置负担。如果需要手动完成该过程，可以重写 rvm_init.c 中的如下两个钩子函数，并在其内部分别创建和初始化各个内核对象。

3.3.1 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 3-2 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。

3.3.2 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 3-3 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。

3.4 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。这些宏定义可以由工程生成器负责填充，也可以由用户手动填充。这些宏定义的列表如下。

表 3-4 原生进程内应当存在的宏定义一览

宏名称	作用
RVM_VIRT_LIB_ENABLE	准虚拟化库的使能宏。在原生进程内它应当被定义为 1。 例子： #define RVM_VIRT_LIB_ENABLE (0U)
RVM_PREEMPT_PRIO_NUM	内核的优先级数量。它应当与 RME 中的同名配置宏维持一致。 例子： #define RVM_PREEMPT_PRIO_NUM (4U)
RVM_PREEMPT_VPRIO_NUM	虚拟机监视器的虚拟优先级数量。它应当与虚拟机监视器配置中的值一致。 例子： #define RVM_PREEMPT_VPRIO_NUM (32U)

宏名称	作用
RVM_KOM_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_KOM_SLOT_ORDER (4U)
RVM_ASSERT_ENABLE	控制断言宏是否默认通过。如果它定义为 0，则不再检查断言。 例子： #define RVM_ASSERT_ENABLE (1U)
RVM_DBGLOG_ENABLE	控制调试打印的开关。如果它定义为 0，则不再调试打印。 例子： #define RVM_DBGLOG_ENABLE (1U)

3.5 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直接在原生进程中使用。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。下面仅对这些系统调用做简单描述；有关这些系统调用的具体信息，请查看 RME 的手册。

3.5.1 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 3-5 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_Cpt_Crt(cid_t Cap_Cpt_Crt, cid_t Cap_Kom, cid_t Cap_Cpt, ptr_t Raddr, ptr_t Entry_Num)
删除权能表	ret_t RVM_Cpt_Del(cid_t Cap_Cpt_Del, cid_t Cap_Del)
传递普通权能	ret_t RVM_Cpt_Add(cid_t Cap_Cpt_Dst, cid_t Cap_Dst, cid_t Cap_Cpt_Src, cid_t Cap_Src, ptr_t Flag) 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_Cpt_Pgt(cid_t Cap_Cpt_Dst, cid_t Cap_Dst, cid_t Cap_Cpt_Src, cid_t Cap_Src, ptr_t Begin, ptr_t End, ptr_t Flag) 备注：该函数不能被用来传递其他类型的权能。Begin 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。该范围包括 End。
传递内核功能	ret_t RVM_Cpt_Kern(cid_t Cap_Cpt_Dst, cid_t Cap_Dst,

名称	接口函数原型
调用权能	cid_t Cap_Cpt_Src, cid_t Cap_Src, ptr_t Begin, ptr_t End) 备注：该函数不能被用来传递其他类型的权能。Begin 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。该范围包括 End。
传递内核内存权能	ret_t RVM_Cpt_Kom(cid_t Cap_Cpt_Dst, cid_t Cap_Dst, cid_t Cap_Cpt_Src, cid_t Cap_Src, ptr_t Begin, ptr_t End, ptr_t Flag) 备注：该函数不能被用来传递其他类型的权能。Begin 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。该范围不包括 End。
权能冻结	ret_t RVM_Cpt_Frz(cid_t Cap_Cpt_Frz, cid_t Cap_Frz)
权能移除	ret_t RVM_Cpt_Rem(cid_t Cap_Cpt_Rem, cid_t Cap_Rem)

3.5.2 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 3-6 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_Kfn_Act(cid_t Cap_Kfn, ptr_t Func_ID, ptr_t Sub_ID, ptr_t Param1, ptr_t Param2)

3.5.3 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 3-7 页表相关系统调用

名称	接口函数原型
页目录创建	ret_t RVM_Pgt_Crt(cid_t Cap_Cpt, cid_t Cap_Kom, cid_t Cap_Pgt, ptr_t Raddr, ptr_t Start_Addr, ptr_t Is_Top, ptr_t Size_Order, ptr_t Num_Order)
删除页目录	ret_t RVM_Pgt_Del(cid_t Cap_Cpt, cid_t Cap_Pgt)
映射内存页	ret_t RVM_Pgt_Add(cid_t Cap_Pgt_Dst, ptr_t Pos_Dst, ptr_t Flag_Dst, cid_t Cap_Pgt_Src, ptr_t Pos_Src, ptr_t Index)
移除内存页	ret_t RVM_Pgt_Rem(cid_t Cap_Pgt, ptr_t Pos)
构造页目录	ret_t RVM_Pgt_Con(cid_t Cap_Pgt_Parent, ptr_t Pos,

名称	接口函数原型
	cid_t Cap_Pgt_Child, ptr_t Flags_Childs)
析构页目录	ret_t RVM_Pgt_Des(cid_t Cap_Pgt_Parent, ptr_t Pos, cid_t Cap_Pgt_Child)

3.5.4 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 3-8 进程相关系统调用

名称	接口函数原型
创建进程	ret_t RVM_Prc_Crt(cid_t Cap_Cpt_Crt, cid_t Cap_Prc, cid_t Cap_Cpt, cid_t Cap_Pgt)
删除进程	ret_t RVM_Prc_Del(cid_t Cap_Cpt, cid_t Cap_Prc)
更改权能表	ret_t RVM_Prc_Cpt(cid_t Cap_Prc, cid_t Cap_Cpt)
更改页表	ret_t RVM_Prc_Pgt(cid_t Cap_Prc, cid_t Cap_Pgt)

3.5.5 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 3-9 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_Cpt, cid_t Cap_Kom, cid_t Cap_Thd, cid_t Cap_Prc, ptr_t Max_Prio, ptr_t Raddr, ptr_t Attr)
创建虚拟机专用线程[1]	ret_t RVM_Hyp_Crt(cid_t Cap_Cpt, cid_t Cap_Kom, cid_t Cap_Thd, cid_t Cap_Prc, ptr_t Max_Prio, ptr_t Raddr, ptr_t Attr)
删除线程	ret_t RVM_Thd_Del(cid_t Cap_Cpt, cid_t Cap_Thd)
设置执行属性	ret_t RVM_Thd_Exec_Set(cid_t Cap_Thd, void* Entry, void* Stack, void* Param)
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)
虚拟机专用线程绑定[2]	ret_t RVM_Hyp_Sched_Bind(cid_t Cap_Thd, cid_t Cap_Thd_Sched, cid_t Cap_Sig, tid_t TID, ptr_t Prio, ptr_t Haddr)
更改优先级	ret_t RVM_Thd_Sched_Prio(cid_t Cap_Thd, ptr_t Prio)

[1] 除非要自行实现虚拟机监视器，否则不要调用

[2] 除非要自行实现虚拟机监视器，否则不要调用

名称	接口函数原型
更改优先级(2)	ret_t RVM_Thd_Sched_Prio2(cid_t Cap_Thd0, ptr_t Prio0, cid_t Cap_Thd1, ptr_t Prio1) 备注：一次可修改两个线程的优先级。
更改优先级(3)	ret_t RVM_Thd_Sched_Prio3(cid_t Cap_Thd0, ptr_t Prio0, cid_t Cap_Thd1, ptr_t Prio1, cid_t Cap_Thd2, ptr_t Prio2) 备注：一次可修改三个线程的优先级。
更改优先级(4)	ret_t RVM_Thd_Sched_Prio4(cid_t Cap_Thd0, ptr_t Prio0, cid_t Cap_Thd1, ptr_t Prio1, cid_t Cap_Thd2, ptr_t Prio2, cid_t Cap_Thd3, ptr_t Prio3) 备注：一次可修改四个线程的优先级。
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 Is_Yield)

3.5.6 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 3-10 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_Cpt, cid_t Cap_Sig)
删除信号端点	ret_t RVM_Sig_Del(cid_t Cap_Cpt, cid_t Cap_Sig)
向端点发送	ret_t RVM_Sig_Snd(cid_t Cap_Sig, ptr_t Number)
从端点接收	ret_t RVM_Sig_Rcv(cid_t Cap_Sig, ptr_t Option)

3.5.7 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 3-11 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_Cpt, cid_t Cap_Kom,

名称	接口函数原型
	cid_t Cap_Inv, cid_t Cap_Prc, ptr_t Raddr)
删除同步迁移调用	ret_t RVM_Inv_Del(cid_t Cap_Cpt, 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 Is_Exc_Ret)
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)

3.5.8 事件源激活相关系统调用

要从原生进程发送消息给虚拟机，可以使用 RVM_Prc_Evt_Snd 函数来激活任何一个事件源，注册了该事件源的虚拟机均会得到一个虚拟中断。由于原生进程本身的优先级高于整个虚拟机子系统[1]，因此本函数并对事件源目标加以限制，原生进程可自行决定发送信号给任意一个事件源。有关事件源，请参见 2.2.7。

表 3-12 激活事件源的所需参数

原型	ret_t RVM_Prc_Evt_Snd(ptr_t Evt_Num)	
参数名称	类型	描述
Evt_Num	ptr_t	事件源的编号。

3.6 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 3-13 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_Int_Print	打印整形数字。
RVM_Hex_Print	打印无符号整形数字。

[1] 这甚至包括了虚拟机监视器中除 SFTD 以外的一切线程

函数	意义
RVM_Str_Print	打印字符串。

3.6.1 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 3-14 变量清空的所需参数

原型	void RVM_Clear(volatile void* Addr, ptr_t Size)	
参数名称	类型	描述
Addr	volatile 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。

3.6.2 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 3-15 创建双向循环链表

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

3.6.3 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 3-16 在双向循环链表中删除节点

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

3.6.4 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 3-17 在双向循环链表中插入节点

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

3.6.5 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。如果不实现该函数，后文的一切调试打印函数都无法使用。工程生成器会在生成的进程文件中提供该函数的原型，用户只需要使用合适的内容来填充它。

表 3-18 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。

3.6.6 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。该函数有一个受调试开关控制的同功能宏 RVM_DBG_I。要使用这个函数，需要 RVM_Putchar 在进程内被声明和实现。

表 3-19 打印整形数字

函数原型	cnt_t RVM_Int_Print(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。

3.6.7 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。该函数有一个受调试开关控制的同功能宏 RVM_DBG_H。要使用这个函数，需要 RVM_Putchar 在进程内被声明和实现。

表 3-20 打印无符号整形数字

函数原型	cnt_t RVM_Hex_Print(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。

3.6.8 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。该函数有一个受调试开关控制的同功能宏 RVM_DBG_S。要使用这个函数，需要 RVM_Putchar 在进程内被声明和实现。

表 3-21 打印字符串

函数原型	cnt_t RVM_Str_Print(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。

3.7 本章参考文献

无

第 4 章 虚拟机

4.1 简介

虚拟机内部可以虚拟化任意的小型 RTOS[1]，并为其提供一个虚拟微控制器环境。该环境的时空和其它虚拟机是互相隔离的，一个虚拟机的崩溃仅会导致自身的重启，而不会导致其它虚拟机的崩溃。这样，故障或安全漏洞在不同组件间的传播就受到限制，使单点失效不至于导致系统整体失效。

虚拟机是一种特殊的 RME 进程，它的一切操作都由 RVM 接管。它的头部也有和原生进程相同的进程描述符，描述其中的所有入口点。然而，虚拟机进程中只有固定的三个入口点，分别是（1）中断向量处理线程 Vct（2）用户程序线程 Usr 和（3）跳板，且不能拥有除了内核功能调用和信号发送端点之外的任何权能。不推荐在虚拟机中调用除信号发送[2]以外的任何 RME 系统调用；原则上一切功能均由 RVM 超级调用接管。

4.2 登记虚拟机到 RVM

在 RVM 中，每个虚拟机的配置被表示成一个 struct RVM_Vmap_Struct 类型的结构体。这些结构体组成一个常量结构体数组 RVM_Vmap，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、状态区地址、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_Struct 组成的数组 RVM_Virt 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。

工程生成器可以完全自动化地进行这个过程，因此我们推荐使用工程生成器建立各个虚拟机。

4.2.1 虚拟机信息的填充

用户应当静态声明结构体数组 const struct RVM_Vmap_Struct RVM_Vmap[]，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 4-1 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 Slice	虚拟机的运行时间片。该值不可以设置为 0。

[1] 如 RMP、FreeRTOS、uC/OS 和 RT-Thread 等

[2] 这是为了和原生进程通信

域名称	作用
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 Vct_Num	虚拟机虚拟中断向量的数目，这个数目不能超过 1920。通常而言任何一个虚拟机都很难用到如此大量的中断源。
struct RVM_Thd_Reg* 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_Thd_Reg 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_State* State_Base	虚拟机的状态缓冲区地址。它指向的内存地址必须不小于 struct RVM_State 结构体的大小。该结构体包括了虚拟机的状态信息，以及其中断向量标志位。它应当与登记在虚拟机进程内部的 RVM_VIRT_STATE_BASE 配置项维持一致。
ptr_t State_Size	虚拟机的状态缓冲区大小，单位是字节。
ptr_t Desc_Base	虚拟机描述头的基地址。它的位置就在虚拟机进程的代码段的开头。
cid_t Vct_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 Vct_Thd_Cap	虚拟机中断向量处理线程的权能号。
ptr_t Vct_Stack_Base	虚拟机中断向量处理线程的栈基址。
ptr_t Vct_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 Usr_Thd_Cap	虚拟机用户程序处理线程的权能号。
ptr_t Usr_Stack_Base	虚拟机用户程序处理线程的栈基址。
ptr_t Usr_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。如果 RTOS 的内核运行中使用这个栈，那么这个栈要设计得至少要足够 RTOS 使用。

工程生成器可以自动生成该数据结构，当然也可以选择手动填充。

4.2.2 创建虚拟机并初始化

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，使中断向量处理线程 Vct 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。

4.2.3 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_Struct RVM_Virt[]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 4-2 虚拟机批量注册钩子

函数原型	void RVM_Virt_Crt(struct RVM_Virt_Struct* Virt, const struct RVM_Vmap_Struct* Vmap, rvm_ptr_t Virt_Num)
返回值	无。
参数	struct RVM_Virt_Struct* Virt 静态声明的虚拟机运行时数据结构体数组的首地址。
	const struct RVM_Vmap_Struct* Vmap 静态声明的虚拟机信息结构体数组的首地址。
	ptr_t Virt_Num 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。

4.3 虚拟机内部应当存在的宏定义

除了那些在原生进程内部必须具备的宏定义之外，虚拟机内部还需要存在如下的配置宏定义。这些宏定义可以由工程生成器负责填充，也可以由用户手动填充。这些宏定义的列表如下。

表 4-3 虚拟机进程内应当存在的宏定义一览

宏名称	作用
RVM_VIRT_LIB_ENABLE	准虚拟化库的使能宏。在虚拟机内它应当被定义为 1。 例子： #define RVM_VIRT_LIB_ENABLE (1U)

宏名称	作用
RVM_VIRT_VCT_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名配置项维持一致。 例子： #define RVM_VIRT_VCT_NUM (10U)
RVM_VIRT_STATE_BASE	虚拟机状态区域的基地址。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_STATE_BASE (0x20005700U)
RVM_VIRT_STATE_SIZE	虚拟机状态区域的大小，单位为字节。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_STATE_SIZE (0x50U)
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 (0x20005400U)
RVM_VIRT_REG_SIZE	用户线程虚拟寄存器组的大小，单位为字节。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_SIZE (0x70U)

4.4 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 4-4 虚拟机可以使用的内部功能函数一览

函数	意义
RVM_Hyp_Putchar	打印单个字符。
RVM_Hyp_Reboot	重启虚拟机。
RVM_Hyp_Int_Ena	开启虚拟总中断。
RVM_Hyp_Int_Dis	关闭虚拟总中断。
RVM_Hyp_Vct_Phys	映射物理中断源到虚拟中断源。
RVM_Hyp_Vct_Evt	映射事件源到虚拟中断源。

函数	意义
RVM_Hyp_Vct_Del	解除虚拟中断源上的映射。
RVM_Hyp_Vct_Lck	锁定虚拟中断源映射。
RVM_Hyp_Vct_Wait	等待虚拟中断源被触发。
RVM_Hyp_Evt_Add	授权向某事件源进行发送。
RVM_Hyp_Evt_Del	撤销向某事件源进行发送的授权。
RVM_Hyp_Evt_Snd	向某事件源进行发送。
RVM_Hyp_Wdg_Clr	激活并投喂看门狗。

4.4.1 打印单个字符

本超级调用会在虚拟机监视器的调试输出中打印一个字符。

表 4-5 打印单个字符

函数原型	void RVM_Hyp_Putchar(char Char)
返回值	无。该调用总是成功。
参数	char Char 要输出到系统控制台的字符。

4.4.2 重启虚拟机

本超级调用会无条件重启虚拟机。

表 4-6 重启虚拟机

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

4.4.3 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 4-7 开启虚拟总中断

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

4.4.4 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 4-8 关闭虚拟总中断

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

4.4.5 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_VIRT_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 4-9 映射物理中断源到虚拟中断源

函数原型	ret_t RVM_Hyp_Vct_Phys(ptr_t Phys_Num, ptr_t Vct_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 Vct_Num 虚拟中断源的中断向量号。	

4.4.6 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 4-10 映射事件源到虚拟中断源

函数原型	ret_t RVM_Hyp_Vct_Evt(ptr_t Evt_Num, ptr_t Vct_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 Vct_Num 虚拟中断源的中断向量号。	

4.4.7 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 4-11 解除虚拟中断源上的映射

函数原型	ret_t RVM_Hyp_Vct_Del(ptr_t Vct_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 Vct_Num 虚拟中断源的中断向量号。	

4.4.8 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 4-12 锁定虚拟中断源映射

函数原型	ret_t RVM_Hyp_Vct_Lck(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。	

4.4.9 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，否则 RVM 会拒绝该超级调用[2]。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[3]。

表 4-13 等待虚拟中断源被触发

函数原型	ret_t RVM_Hyp_Vct_Wait(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。	

[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

[2] 若不拒绝该超级调用，虚拟机一旦进入休眠就无法唤醒；在真实的处理器上，先关中断再休眠同样会导致死机

[3] 如 ARMv7-M 的 WFE 和 WFI

4.4.10 授予向某事件源进行发送的权限

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 4-14 授予向某事件源进行发送的权限

函数原型	ret_t RVM_Hyp_Evt_Add(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。	

4.4.11 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 4-15 撤销向某事件源进行发送的授权

函数原型	ret_t RVM_Hyp_Evt_Del(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。	

4.4.12 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 4-16 向某事件源进行发送

函数原型	ret_t RVM_Hyp_Evt_Snd(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_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 要激活的事件源的编号。	

4.4.13 激活并投喂看门狗

当虚拟机信息中的 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[1]，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 4-17 激活并投喂看门狗

函数原型	ret_t RVM_Hyp_Wdg_Clr(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。	

4.5 虚拟机可以使用的内部功能函数

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 4-18 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_Vct_Loop	运行虚拟中断向量处理循环。
RVM_Virt_Int_Mask	屏蔽虚拟中断向量处理。
RVM_Virt_Int_Unmask	解除虚拟中断向量处理屏蔽。
RVM_Virt_Vct_Reg	注册普通虚拟中断向量处理函数。

[1] 看门狗只计算虚拟机本身的运行时间，而非真实系统的运行时间

函数	意义
RVM_Virt_Tim_Reg	注册时钟虚拟中断向量处理函数
RVM_Virt_Ctx_Reg	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。

4.5.1 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。该操作应该由中断处理线程 Vct 在虚拟机启动时最先调用。在使用工程生成器的场合，工程生成器会在生成的 Vct 线程中调用本函数。

表 4-19 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。

4.5.2 运行虚拟中断向量处理循环

本操作循环处理虚拟机内部的中断向量，一经调用永不返回。该操作应该由中断处理线程 Vct 在最后调用。在使用工程生成器的场合，工程生成器会在生成的 Vct 线程中调用本函数。

表 4-20 运行虚拟中断向量处理循环

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

4.5.3 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，因此用户程序线程 Usr 仍然会被中断处理线程 Vct 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。相比关闭虚拟总中断的 RVM_Hyp_Int_Dis 超级调用，本操作无需唤起虚拟机监视器，因此高效得多，非常适合用于 RTOS 锁调度器的场合。本操作没有嵌套计数，因此需要客户机 RTOS 自行进行嵌套计数。

表 4-21 屏蔽虚拟中断向量处理

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

4.5.4 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，是 RVM_Virt_Int_Mask 的逆操作。在解除屏蔽后，如果发现在中断屏蔽期间有虚拟中断悬起[1]，则会立刻触发 Vct 线程的运行来处理它们。本操作没有嵌套计数，因此需要客户机 RTOS 自行进行嵌套计数。

表 4-22 解除虚拟中断向量处理屏蔽

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

4.5.5 注册普通虚拟中断向量

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；如果一个未注册中断向量处理函数的虚拟中断被激活，则该中断将被直接略过。

表 4-23 注册普通虚拟中断向量

函数原型	ret_t RVM_Virt_Vct_Reg(ptr_t Vct_Num, void (*Vct)(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_RANGE 传入的虚拟中断源的中断向量号越界。
参数	ptr_t Vct_Num 虚拟中断源的中断向量号。 void (*Vct)(void) 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[2]，那么意味着解除对该中断向量的注册。

4.5.6 注册时钟虚拟中断向量

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。客户机 RTOS 在初始化序列中应当通过本操作注册其时钟虚拟中断向量。

表 4-24 注册时钟虚拟中断向量

函数原型	void RVM_Virt_Tim_Reg(void (*Tim)(void))

[1] 已被触发但尚未被处理

[2] 空指针，也即 NULL

返回值	无。该操作总是成功。
参数	void (*Tim)(void) 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[1]，那么意味着解除对时钟中断向量的注册。

4.5.7 注册线程切换虚拟中断向量

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。客户机 RTOS 在初始化序列中应当通过本操作注册其线程切换虚拟中断向量。

表 4-25 注册线程切换虚拟中断向量

函数原型	void void RVM_Virt_Ctx_Reg(void (*Ctx)(void))
返回值	无。该操作总是成功。
参数	void (*Ctx)(void) 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[2]，那么意味着解除对线程切换中断向量的注册。

4.5.8 触发线程切换虚拟中断向量

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 4-26 触发线程切换虚拟中断向量

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

4.6 本章参考文献

无

[1] 空指针，也即 NULL

[2] 空指针，也即 NULL

第 5 章 工程生成器

5.1 简介

工程编译器负责根据工程描述文件生成所需的启动文件、链接器脚本和构建系统，免去手动配置系统的繁琐。在内部，它分为输入分析模块、工程生成模块和工程输出模块三个部分。它可以通过命令行界面调用，也可以通过图形前端调用。

5.1.1 输入分析模块

输入分析模块负责分析各个描述文件并将其打包成内部数据结构传递给工程生成模块。它会进行以下工作：

（1）分析工程描述文件（*.rvp），

（2）根据工程描述文件读取合适的平台描述文件（*.rva），

（3）根据工程描述文件读取合适的芯片描述文件（*.rvc），

（4）检查各描述文件的合法性，以及

（5）加载工程描述文件对应的架构平台、编译工具链、构建系统与客户机操作系统的相关生成工具。

5.1.2 工程生成模块

工程生成模块负责读取输入分析模块提供工程信息、平台信息和芯片信息，生成工程所需的各项数据，并将这些数据交由工程输出模块生成真正的工程。它会进行以下工作：

（1）对齐工程中声明的内存块，

（2）为工程中声明的内存块分配实际基地址，

（3）将共享内存添加到各个进程的内存布局，

（4）为各个进程分配页表，

（5）为各个内核对象分配权能编号，

（6）决定内核对象的权能托管（授权）布局，以及

（7）为各个内核对象分配内核内存。

5.1.3 工程输出模块

工程输出模块负责读取工程生成模块的输出，并且生成实际工程文件。它会进行以下工作：

（1）生成内核配置头文件（rme_platform_conf.h 及 rme_platform_xxx.h），

（2）生成内核启动头文件（rme_boot.h），

（3）生成内核启动源文件（rme_boot.c），

（4）生成内核钩子源文件（rme_hook.c），

（5）生成内核中断向量源文件（rme_handler_xxx.c），

（6）生成内核链接器脚本（rme_platform_xxx.ld 等），

（7）生成内核构建工程文件（makefile 等），

（8）生成虚拟机监视器配置头文件（rvm_platform_conf.h 及 rvm_platform_xxx.h），

（9）生成虚拟机监视器启动头文件（rvm_boot.h），

（10）生成虚拟机监视器启动源文件（rvm_boot.c），

（11）生成虚拟机监视器钩子源文件（rvm_hook.c），

（12）生成虚拟机监视器链接器脚本（rvm_platform_xxx.ld 等），

（13）生成虚拟机监视器构建工程文件（makefile 等），

（14）生成各进程的主要头文件（prc_xxx.h），

（15）生成各进程的入口点源文件（prc_xxx_thd_xxx.c 及 prc_xxx_inv_xxx.c），

（16）生成各进程的描述符源文件（prc_xxx_desc.c），

（17）生成各个进程的主要源文件（prc_xxx.c），

（18）如果进程是虚拟机，生成虚拟机配置头文件（随客户机操作系统不同而不同），

（19）如果进程是虚拟机，生成虚拟机配置源文件（随客户机操作系统不同而不同），

（20）生成各进程链接器脚本（prc_xxx.ld 等），

（21）生成各进程构建工程文件（makefile 等），以及

（22）生成包含以上各工程文件的工作空间（如果适用的话）。

5.1.4 图形前端模块

由于工程描述文件内部结构也较为繁琐，手动编辑容易引发错误，工程生成器还提供了用来编辑工程的图形前端。该图形前端能够让工程编辑变得更加直观便捷。

5.2 命令行参数描述

工程生成器的命令行参数仅包括一项-i，它指定了工程描述文件的路径。工程描述文件的后缀为*.rvp，内部包含了生成工程所需的其它一切信息。

例如：-i path/to/input.rvp

5.3 工程描述文件结构<Project>

工程主文件记载了关于该工程的一切信息，其后缀名为*.rvp；其内部的的所有标签都位于一个<Project>大标签下。

所有标签，只要下述描述中有，不论其内容有无，均不可省略。

5.3.1 工程名称<Name>

该配置项记录了工程的名称。通常而言，它与工程文件的名称是一致的。

工程生成器不关心该配置项的值。

5.3.2 工程版本<Version>

该配置项记录了工程的版本。

工程生成器不关心该配置项的值。

5.3.3 启用断言<Assert_Enable>

该配置项决定了是否检查代码中的断言，可以是 Yes 或 No。如果假设所有断言均正确而不加检查，可以选择 No；这会略微提升运行速度并减小代码体积。

5.3.4 启用调试打印<Debug_Log_Enable>

该配置项决定了是否以启用 RVM_DBG_开头的调试打印宏，可以是 Yes 或 No。如果不需要调试打印输出，可以选择 No；这会略微提升运行速度并减小代码体积。

5.3.5 启用固定式裸页表<Pgtbl_Raw_Enable>

该配置项决定了是否启用固定的用户态 MPU 元数据，可以是 Yes 或 No。一旦启用，每个进程的、包含了其一切地址空间信息的 MPU 元数据都将提前一次性生成并在进程创建时加载给内核，内核中涉及页表的系统调用和动态页功能将不再编译，页表内核对象在启动时也不再创建。这在那些空间吃紧，且不需要动态页提供的额外灵活性的场景中非常有效。

5.3.6 工作空间构建系统<Buildsystem>

该配置项记录了工作空间使用的构建系统。部分构建工具[1]允许将多个工程组成一个工作空间，在这里选择它的类别。当然，也可以选择“None”，此时不输出任何工作空间。

5.3.7 工作空间输出目录<Workspace_Output>

该配置项是一个相对路径，记载了工作空间的输出目录。

5.3.8 工作空间输出覆盖<Workspace_Overwrite>

该配置项指定了工作空间文件是否会在重新生成工程时被覆盖，可以是 Yes 或 No。有时我们已经对工作空间文件进行了一定的手动修改，不希望它被覆盖，此时可以选择 No。

5.3.9 芯片配置<Chip>

本配置组负责架构平台和芯片型号的选择。工程编译器将从此配置项中抽取信息决定读取的平台描述文件（*.rva）和芯片描述文件（*.rvc）。

[1] 如 Keil uVision 允许使用*.uvmpw 工作空间包含多个*.uvprojx 工程文件

5.3.9.1 目标平台<Platform>

该配置项记录了工程的架构平台。每个平台用一个代号代表，比如 A7M 对应于 ARMv7-M 平台。每个平台都对应一个平台描述文件，它们位于 Include/Platform/XXX/rvm_platform_xxx.rva。有关这些描述文件的结构，请参见 5.4。

5.3.9.2 芯片类别<Class>

该配置项记录了具体的芯片类别。该类别仅仅决定芯片的内部计算资源配置，并不指定到具体封装，比如 STM32F767IG 可以兼容 STM32F767IGT6 和 STM32F767IGK6。每个芯片类别都对应一个芯片描述文件，它们位于 Include/Platform/XXX/Chip/rvm_platform_xxx.rvc。有关这些描述文件的结构，请参见 5.5。

5.3.9.3 芯片名称<Name>

该配置项记录了芯片的精确全名，如 STM32F767IGT6。工程生成器将检查该全名是否在芯片类别指定的兼容芯片中。如果不在，则工程生成器将报错。

5.3.9.4 启用协处理器<Coprocessor>

该配置项是一个逗号分隔列表，它记录了项目中需要启用的协处理器，如 FPU 等，只有在这里启用的协处理器才会被内核初始化。工程生成器将检查该协处理器是否在芯片中存在。如果不在，则工程生成器将报错。

5.3.9.5 初始化配置<Config>

初始化配置组包括了架构的初始化参数和芯片的初始化参数；这些参数的存储方式都是<标签>值</标签>。这些参数是依架构和芯片而异的；平台描述文件和芯片描述文件中声明的全部参数在这里都必须一个不多一个不少地填写。

5.3.10 额外存储器<Extmem><En>

本配置组包含了要添加进系统的额外内存块的属性。额外内存是指微控制器可以访问的外部存储器，比如外扩的 SDRAM、SRAM 等等。如果需要工程生成器调用它们，那么就必须将它们的信息登记在该组内部。此时，总可用存储器就是芯片固有存储器[1]与此处所列的额外存储器的总和，共享存储块[2]和进程私有存储块将从总可用存储器分配。此外，还必须在内核启动钩子中编写合适的外扩存储器初始化代码[3]，以确保这些存储器可以正确访问。

---

[1] 请参见 5.5.9

[2] 请参见 5.3.10

[3] 如 STM32、GD32、HC32 等微控制器具备的 FSMC、FMC 或 EMC 等

5.3.10.1 额外段基址<Base>

该配置项是一个十六进制值，决定了该外部内存段的起始地址。

5.3.10.2 额外段长度<Size>

该配置项是一个十六进制值，决定了该外部内存段的长度。

5.3.10.3 额外段类别<Type>

该配置项决定了存储段的类别。可以选择的类别有三种：代码（Code）、数据（Data）、设备（Device）。工程生成器在分配内存时，会使用这个类别决定将该段存储器用于何种用途。

5.3.10.4 额外段属性<Attribute>

该配置项决定了内存段的属性。内存段属性分别有可读（Readable，R）、可写（Writable，W）、可执行（eXecutable，X）、可缓冲（Bufferable，B）、可缓存（Cacheable，C）、静态映射（Static，S）六种，它们可以以任意组合进行叠加。工程生成器在分配存储器时，仅会安排属性是此属性子集的存储段到该物理存储段上。比如，用户若指定某额外存储段的属性不包括可写[1]，那么工程生成器将绝不会将可写内容分配在该存储段上。

一般地，代码段 ROM 的属性应设置为 RXBCS，数据段 RAM 的属性应设置为 RWBCS。设备段的属性则应设置为 RWS。

5.3.11 共享存储块<Shmem><Sn>

本配置组包含了系统中存在的共享存储块的属性。共享存储块是指可以被多个进程或虚拟机访问的存储区域，比如数据缓冲区等等；它类似于 Linux 等系统中的共享内存，可用于多个进程或虚拟机间的受控信息传递。本配置组中声明的共享存储块会从总可用存储器[2]中分配，然后被添加进每一个引用它的进程或虚拟机的页表中。

5.3.11.1 共享段名称<Name>

该配置项是一个必填的字段，要求用户输入该共享内存段的名称。该名称必须是合法的标识符，并且必须独一无二。

5.3.11.2 共享段基址<Base>

该配置项是一个可选的十六进制值。用户应该填入该共享内存段的起始地址。如果填写“Auto”，则代表该共享内存段的位置由编译器自动分配。

---

[1] 比如外挂 XIP SPI Flash

[2] 请参见 5.3.9

5.3.11.3 共享段长度<Size>

该配置项是一个必填的十六进制值。用户应该填入该共享内存段的长度。

5.3.11.4 共享段对齐方式<Align>

该配置项是一个必填的十进制值，决定了该存储器的对齐方式。

当段基址为“Auto”时，如果填写为 N，则该段存储器在自动分配时对齐到 2^N；如果填写为“Auto”，则代表该共享内存段的对齐方式由编译器自主决定。

当段基址是固定值时，该配置项必须填写“Auto”。

对于同一段内存，不同的架构支持的对齐方式可能有不同。

5.3.11.5 共享段类别<Type>

该配置项是决定了内存段的类别。可以选择的类别有三种：代码（Code）、数据（Data）、设备（Device）。编译器在分配内存时，会使用这个类别决定将该共享内存段分配到哪一种存储器。

5.3.11.6 共享段属性<Attribute>

该配置项决定了内存段的属性。内存段属性分别有可读（Readable，R）、可写（Writable，W）、可执行（eXecutable，X）、可缓冲（Bufferable，B）、可缓存（Cacheable，C）、静态映射（Static，S）六种，它们可以以任意组合进行叠加。编译器在检查在各个进程或虚拟机中声明的对共享内存的引用时，仅会允许这个属性的子集出现。比如，用户若指定该共享内存段的属性不包括可写，那么编译器会检查并确保后续的对共享内存的引用不包括写属性。

5.3.12 内核配置<Kernel>

本配置组负责 RME 内核的基本配置，包括内核使用的存储器、内核支持的优先级，以及内核工程包含的各文件的存储位置等。

5.3.12.1 代码段基址<Code_Base>

该配置项决定了内核所使用的代码段的基址。

5.3.12.2 代码段长度<Code_Size>

该配置项决定了内核所使用的代码段的长度。内核的代码段包括了内核的汇编代码以及其只读数据区，通常在某个架构上是一个固定值。

5.3.12.3 数据段基址<Data_Base>

该配置项决定了内核所使用的数据段的基址。

5.3.12.4 数据段长度<Data_Size>

该配置项决定了内核所使用的数据段的长度。内核的数据段包括了内核的可读写数据区、内核对象用内存区和内核栈区，通常要根据创建的内核对象的多寡进行调整。如果生成器报告内核数据内存耗尽而无法分配内核对象，那么可以尝试将这个值修改的大一些。

5.3.12.5 内核栈长度<Stack_Size>

该配置项决定了内核所使用的内核栈段的长度。作为微控制器上运行的微内核，1kB 的内核栈对于大多数用途都足够了。

5.3.12.6 额外内核对象用内存长度<Extra_Kom>

该配置项决定了在系统创建完所有与生成器生成的代码相关的的内核对象后，还额外需要保留给用户用来创建内核对象的额外内核内存长度。它只有在用户要手动在生成器生成的 RVM 启动代码中添加内核对象的创建代码时才需要设置为不为 0 的数字。通常而言，我们不推荐用户手动创建内核对象，这意味着用户有可能破坏系统的信息安全性。

5.3.12.7 内核内存分配粒度<Kom_Order>

该配置项决定了系统的内核内存分配粒度，可在 8 字节，16 字节和 32 字节这三项中选一个最适合的。系统在创建内核对象时会将所有的内核内存都对齐到这一粒度。如果系统本身的内存比较少，推荐使用 8 字节对齐分配以减少碎片；如果系统本身的内存较多，则可以采用 32 字节分配以降低内存分配登记表本身的大小。在默认状况下，设置为 16 字节分配即可。

5.3.12.8 优先级数量<Kern_Prio>

该配置项决定了系统中内核态优先级的数量，应该填入一个是处理器字长的整数倍的十进制数值。通常而言，只要填入处理器字长就足够了，比如是 32 位处理器则填写 32。

5.3.12.9 构建系统<Buildsystem>

决定内核工程采取何构建系统。

5.3.12.10 构建工具链<Toolchain>

决定内核工程采取何构建工具链。

5.3.12.11 优化等级<Optimization>

决定内核工程所使用的优化等级。可选择的优化等级有不优化（O0）、基本优化（O1）、全面优化（O2）、极限优化（O3）、速度优化（Of）和体积优化（Os）。

5.3.12.12 内核根目录<Kernel_Root>

决定 RME 内核源码根文件夹的位置，是一个相对于工程描述文件的路径。生成的内核工程将引用这里的存放的 RME 内核文件。

5.3.12.13 工程输出目录<Project_Output>

决定内核工程的输出目录，是一个相对于工程描述文件的路径。

5.3.12.14 工程覆盖<Project_Overwrite>

决定在重新生成时是否覆盖之前生成的内核工程文件[1]，可以是 Yes 或 No。如果用户手动编辑了工程而希望保留这些更改，可以选择 No 以保留原来的老版本。

5.3.12.15 链接器脚本输出目录<Linker_Output>

决定内核工程链接器脚本 rme_platform_xxx.ld[2]的输出目录，是一个相对于工程描述文件的路径。内核链接器脚本在每次生成中总是会被覆盖。

5.3.12.16 配置头文件输出目录<Config_Header_Output>

决定内核配置头文件 rme_platform_conf.h 和 rme_platform_xxx.h 的输出目录，是一个相对于工程描述文件的路径。内核配置头文件在每次生成中总是会被覆盖。

5.3.12.17 启动头文件输出目录<Boot_Header_Output>

决定内核启动用头文件 rme_boot.h 的输出目录，是一个相对于工程描述文件的路径。内核启动头文件在每次生成中总是会被覆盖。

5.3.12.18 启动源文件输出目录<Boot_Source_Output>

决定内核启动用源文件 rme_boot.c 的输出目录，是一个相对于工程描述文件的路径。它在每次生成中总是会被覆盖。在该源文件中会完成对内核中断向量信号端点的创建操作。

5.3.12.19 钩子源文件输出目录<Hook_Source_Output>

决定内核钩子源文件 rme_hook.c 的输出目录，是一个相对于工程描述文件的路径。该文件内部包含四个钩子函数，可以利用它在内核初始化流程中插入一些操作。其中：

（1）RME_Boot_Pre_Init 钩子会在内核初始化数据结构前被调用，用户可在其中完成内核启动前必须完成的软硬件初始化工作。

---

[1] 如 Keil uVision 的*.uvprojx 等

[2] 随选择的工具链不同，这里的后缀名也可能不同

（2）RME_Boot_Post_Init 钩子会在内核进入用户态执行前被调用，用户可在其中完成额外的内核对象定制创建工作。启动权能表[1]中保留的额外项数由<Monitor><Extra_Captbl>决定，保留的额外内核内存则由<Kernel><Extra_Kom>决定。其中，可用的权能表项从 RME_RVM_CAP_DONE_FRONT 开始直到 RME_RVM_INIT_CPT_SIZE-1，而可用的内核内存从 RME_RVM_KOM_DONE_FRONT 开始直到 RME_KOM_VA_SIZE-1。需要注意的是，这些额外保留资源是和 RVM 的启动时钩子共享的[2]，如果选择在 RME 的钩子中使用这些资源，在 RVM 的钩子中就不能再次使用它们了。当然，也可以将其中一部分资源用来在 RME 启动时创建内核对象，另一部分资源则留到 RVM 启动时来创建内核对象；此时两部分资源的划分需要用户自行手动完成。

（3）RME_Reboot_Failsafe 钩子会在内核发生严重故障[3]时被调用，用户可在其中完成相应的安全处理工作。

（4）RME_Hook_Kfn_Handler 钩子会在内核遇到未预定义的内核功能调用时被调用，用户可在其中定制额外的内核功能调用。

5.3.12.20 钩子源文件覆盖<Hook_Source_Overwrite>

决定在重新生成时是否覆盖之前生成的内核钩子源文件，可以是 Yes 或 No。如果用户手动编辑了钩子源文件且希望保留更改，可以选择 No 以保留原来的老版本。

5.3.12.21 中断向量源文件输出目录<Handler_Source_Output>

决定内核中断向量上半部源文件 rme_handler_xxx.c 的输出目录，是一个相对于工程描述文件的路径。每个物理中断向量都对应一个独立的源文件；用户可以编辑其中的 RME_Vct_XXX_Handler 函数以在进入物理中断时做一些工作，比如实现中断的上半部处理[4]。这个函数允许用户返回一个返回值，决定中断上半部结束后如何发送信号：（1）若返回 RME_RVM_VCT_SIG_ALL 则向自身的专用端点和 RME_BOOT_INIT_VCT 端点都发送信号，（2）若返回 RME_RVM_VCT_SIG_SELF 则仅向自身的专用端点发送信号，（3）若返回 RME_RVM_VCT_SIG_INIT 则仅向 RME_BOOT_INIT_VCT 端点发送信号，（4）若返回 RME_RVM_VCT_SIG_NONE 则不向任何端点发送信号。

5.3.12.22 中断向量源文件覆盖<Handler_Source_Overwrite>

决定在重新生成时是否覆盖之前生成的中断向量源文件。如果用户手动编辑了中断向量源文件且希望保留更改，可以选择 No 以保留原来的老版本。

5.3.13 监视器配置<Monitor>

[1] 也即 RVM 虚拟机监视器进程使用的权能表；它是系统中第一个权能表

[2] 对应宏的值也是完全相同的；关于这些宏请参见 7.2.2

[3] 如断言检查失败

[4] 上半部-下半部二分法是非常常见的一种中断处理方法：它将中断处理中那些最紧要的部分放在内核中执行，并将那些不甚紧要的部分传递到用户态执行。

本配置组负责 RVM 虚拟机监视器的基本配置，包括 RVM 使用的存储器、支持的虚拟优先级数目、支持的事件源数目及虚拟中断映射数目，以及 RVM 工程包含的各文件的存储位置等。

5.3.13.1 代码段长度<Code_Size>

该配置项是决定了 RVM 所使用的代码段的长度。RVM 的代码段包括了其汇编代码以及其只读数据区，通常在某个架构上是一个固定值。需要注意的是，RVM 的代码起始地址永远直接位于内核代码段后面，因此代码内存基址是自动决定好的，无需用户手动设置。

5.3.13.2 数据段长度<Data_Size>

该配置项决定了 RVM 所使用的数据段的长度。RVM 的数据段包括了其可读写数据区和栈区，通常要根据是否开启虚拟机功能以及虚拟机的数量决定。如果在编译 RVM 时提示内存不够，那么可以尝试将这个值修改的大一些。需要注意的是，RVM 的数据段永远直接位于内核数据段后，因此数据段基址是自动决定好的，无需用户手动设置。

5.3.13.3 启动守护线程栈长度<Init_Stack_Size>

该配置项决定了启动守护线程 INIT 的栈长度。通常而言，256B 是足够的。

5.3.13.4 安全守护线程栈长度<Sftd_Stack_Size>

该配置项决定了安全守护线程 SFTD 的栈长度。通常而言，256B 是足够的。

5.3.13.5 虚拟机守护线程栈长度<Vmmd_Stack_Size>

该配置项决定了虚拟机守护线程 VMMD 的栈长度。通常而言，512B 是足够的。该线程将同时用于超级调用守护协程 HYPD、向量守护协程 VCTD 和时间守护协程 TIMD 的协同运行。

5.3.13.6 额外权能表槽位数<Extra_Captbl>

该配置项决定了系统创建完所有与生成器生成的代码相关的内核对象后，还额外需要保留给用户用来创建内核对象的额外权能表槽位数。它只有在用户要手动在生成器生成的 RVM 启动代码中添加内核对象的创建代码时才需要设置为不为 0 的数字。通常而言，我们不推荐用户手动创建内核对象，这意味着用户有可能破坏系统的信息安全性。

5.3.13.7 启用空闲休眠<Idle_Sleep_Enable>

该配置项决定了在没有原生进程和虚拟机可供运行时是否使处理器进入休眠状态，可以是 Yes 或 No。如果不希望进入低功耗休眠状态以尽量降低中断延迟，可以选择 Yes。

5.3.13.8 虚拟化优先级数<Virt_Prio>

该配置项决定了虚拟机监视器支持的优先级的数量。该优先级数量仅对虚拟机生效，而且必须是处理器字长的整数倍。通常而言，填充为处理器字长，如 32 位处理器则填充 32 就可以了。如果不使用虚拟机的话，这里可以填充为 0。

5.3.13.9 虚拟机事件源数<Virt_Event>

该配置项决定了虚拟机监视器支持的事件源的数量，它必须是处理器字长的整数倍，而且在使用虚拟机的情况下不能为 0。同时，事件源的数量不能超过 1024 的硬性限制。事件源是用于从原生进程到虚拟机以及虚拟机本身之间的沟通的，它可以被映射到虚拟机的虚拟中断向量。

5.3.13.10 虚拟机向量映射数<Virt_Map>

该配置项决定了事件源或物理中断源到虚拟中断向量映射的总数目，它必须大于事件源的数量，而且在使用虚拟机的情况下不能为 0。一个事件源或物理中断源可以被映射到多个虚拟向量，每一次从具体源到具体虚拟向量的映射都会消耗一个映射数。

5.3.13.11 构建系统<Buildsystem>

决定虚拟机监视器工程采取何构建系统。

5.3.13.12 构建工具链<Toolchain>

决定虚拟机监视器工程采取何构建工具链。

5.3.13.13 优化等级<Optimization>

决定虚拟机监视器工程所使用的优化等级。可选择的优化等级有不优化（O0）、基本优化（O1）、全面优化（O2）、极限优化（O3）、速度优化（Of）和体积优化（Os）。

5.3.13.14 虚拟机监视器根目录<Monitor_Root>

决定 RVM 虚拟机监视器源码根文件夹的位置，是一个相对于工程描述文件的路径。生成的虚拟机监视器工程将以相对路径引用这里的虚拟机监视器源文件。

5.3.13.15 工程输出目录<Project_Output>

决定 RVM 工程的输出目录，是一个相对于工程描述文件的路径。

5.3.13.16 工程覆盖<Project_Overwrite>

决定在重新生成时是否覆盖之前生成的虚拟机监视器工程文件[1]，可以是 Yes 或 No。如果用户手动编辑了工程而希望保留这些更改，可以选择 No 以保留原来的老版本。

5.3.13.17 链接器脚本输出目录<Linker_Output>

决定虚拟机监视器工程链接器脚本 rvm_platform_xxx.ld[2]的输出目录，是一个相对于工程描述文件的路径。链接器脚本在每次生成中总是会被覆盖。

5.3.13.18 配置头文件输出目录<Config_Header_Output>

决定虚拟机监视器配置头文件 rvm_platform_conf.h 和 rvm_platform_xxx.h 的输出目录，是一个相对于工程描述文件的路径。配置头文件在每次生成中总是会被覆盖。

5.3.13.19 启动头文件输出目录<Boot_Header_Output>

决定虚拟机监视器启动头文件 rvm_boot.h 的输出目录，是一个相对于工程描述文件的路径。启动头文件在每次生成中总是会被覆盖。

5.3.13.20 启动源文件输出目录<Boot_Source_Output>

决定虚拟机监视器启动源文件 rvm_boot.c 的输出目录，是一个相对于工程描述文件的路径。启动源文件在每次生成中总是会被覆盖。

5.3.13.21 钩子源文件输出目录<Hook_Source_Output>

决定虚拟机监视器钩子源文件 rvm_hook.c 的输出目录，是一个相对于工程描述文件的路径。该文件是用户可编辑的，可以利用它在虚拟机监视器启动之前以及启动之后做一些硬件初始化操作。其中：

（1）RVM_Boot_Pre_Init 钩子会在虚拟机监视器初始化数据结构前被调用，用户可在其中完成虚拟机监视器启动前必须完成的软硬件初始化工作。

（2）RVM_Boot_Post_Init 钩子会在虚拟机监视器初始化结束后被调用，用户可在其中完成额外的内核对象定制创建工作。启动权能表[3]中保留的额外项数由<Monitor><Extra_Captbl>决定，保留的额外内核内存则由<Kernel><Extra_Kom>决定。其中，可用的权能表项从 RVM_CAP_DONE_FRONT 开始直到 RVM_BOOT_CPT_SIZE-1，而可用的内核内存从 RVM_KOM_DONE_FRONT 开始直到 RVM_KOM_VA_SIZE-1。需要注意的是，这些额外保留资源是和 RME 的启动时钩子共享的[4]，如果选择在 RME 的钩子中使用这些资源，在 RVM 的钩子中就不能再次使用它们了。当然，也可以将其中一部分

[1] 如 Keil uVision 的*.uvprojx 等

[2] 随选择的工具链不同，这里的后缀名也可能不同

[3] 也即 RVM 虚拟机监视器进程使用的权能表；它是系统中第一个权能表

[4] 对应宏的值也是完全相同的；关于这些宏请参见 7.2.2

资源用来在 RME 启动时创建内核对象，另一部分资源则留到 RVM 启动时来创建内核对象；此时两部分资源的划分需要用户自行手动完成。

5.3.13.22 钩子源文件覆盖<Hook_Source_Overwrite>

决定在重新生成时是否覆盖之前生成的虚拟机监视器钩子源文件，可以是 Yes 或 No。如果用户手动编辑了钩子源文件且希望保留更改，可以选择 No 以保留原来的老版本。

5.3.14 原生进程配置<Process><Pn>

本配置组负责系统中的原生进程的配置。系统中可以有一个或多个原生进程，因此该类可以有一个或多个实例。这些实例均会被显示在左侧第二栏中。每一个实例都有十个配置组，它们分别占据一个选项卡。

5.3.14.1 进程名称<Name>

该配置项是一个必填的字段，决定进程的名称。该名称必须是一个合法的标识符，并且和系统中的其他进程、虚拟机的名称不得重复。

5.3.14.2 额外权能表槽位数<Extra_Captbl>

该配置项是一个必填的整数值。用户应该填入额外需要保留给用户使用的该进程的权能表槽位数。它只有在用户要手动添加内核对象到该进程的权能表时才需要设置为不为 0 的数字。通常而言，我们不推荐用户手动创建内核对象，这意味着用户有可能破坏系统的信息安全性。

5.3.14.3 启用协处理器<Coprocessor>

该配置项是一个逗号分隔列表，它记录了进程工程中需要启用的协处理器，如 FPU 等，只有在这里启用的协处理器才会被内核初始化。工程生成器将检查该协处理器是否在在项目中被启用。如果不在，则工程生成器将报错。

5.3.14.4 构建系统<Buildsystem>

决定进程工程采取何构建系统。

5.3.14.5 构建工具链<Toolchain>

决定进程工程采取何构建工具链。

5.3.14.6 优化等级<Optimization>

决定进程工程所使用的优化等级。可选择的优化等级有不优化（O0）、基本优化（O1）、全面优化（O2）、极限优化（O3）、速度优化（Of）和体积优化（Os）。

5.3.14.7 工程输出目录<Project_Output>

决定进程工程的输出目录，是一个相对于工程描述文件的路径。

5.3.14.8 工程覆盖<Project_Overwrite>

决定在重新生成时是否覆盖之前生成的进程工程文件[1]，可以是 Yes 或 No。如果用户手动编辑了工程而希望保留这些更改，可以选择 No 以保留原来的老版本。

5.3.14.9 链接器脚本输出目录<Linker_Output>

决定进程工程链接器脚本 prc_xxx_xxx.ld[2]的输出目录，是一个相对于工程描述文件的路径。链接器脚本在每次生成中总是会被覆盖。

5.3.14.10 主头文件输出目录<Main_Source_Output>

决定进程主头文件 prc_xxx.h 的输出目录，是一个相对于工程描述文件的路径。主头文件在每次生成中总是会被覆盖。

5.3.14.11 主源文件输出目录<Main_Source_Output>

决定进程主源文件 prc_xxx.c 以及描述符文件 prc_xxx_desc.c 的输出目录，是一个相对于工程描述文件的路径。主源文件和描述符文件在每次生成中总是会被覆盖；主源文件内含一个用户可填充的调试打印函数 RVM_Putchar，如果用户填充它且使能了调试打印，则可以使用所有 RVM_DBG_开头的调试打印宏。

5.3.14.12 入口点源文件输出目录<Entry_Source_Output>

决定进程中线程与迁移调用入口点源文件 prc_xxx_thd_xxx.c 和 prc_xxx_inv_xxx.c 的输出目录，是一个相对于工程描述文件的路径。这些文件与进程中定义的线程和迁移调用入口是一一对应的；修改这些文件即可实现接管相关线程或迁移调用的执行。其中，原生线程应当永不返回，而迁移调用在结尾处必须调用 RVM_Inv_Ret 主动返回。

此配置项仅适用于原生进程，对虚拟机不存在。

5.3.14.13 入口点源文件覆盖<Entry_Source_Overwrite>

决定在重新生成时是否覆盖之前生成的进程入口点源文件，可以是 Yes 或 No。如果用户手动编辑了入口点源文件而希望保留这些更改，可以选择 No 以保留原来的老版本。

此配置项仅适用于原生进程，对虚拟机不存在。

---

[1] 如 Keil uVision 的*.uvprojx 等

[2] 随选择的工具链不同，这里的后缀名也可能不同

5.3.14.14 私有存储块<Memory><Mn>

本配置组包含了该进程中存在的私有存储块的属性。该组中至少要包含一块代码（Code）存储块和一块数据（Data）存储块；声明的第一块代码存储块将存放进程的全部代码，声明的第一块数据存储块将来存放进程的全部数据区和栈区。其余的代码存储块和数据存储块通常用来存放共享的代码和数据，可通过指针形式使用。

5.3.14.14.1 私有段名称<Name>

该配置项是一个选填的字段，可决定该私有内存段的名称。该名称必须是合法 C 标识符，并且在一个进程内部的内存段之间必须不分大小写地不同。工程生成器和 RVM 本身并不使用该名称，但若用户提供该名称，在进程的主头文件 prc_xxx.h 中将生成各内存段的基址与长度的宏定义，方便用户编程：代码段的基址和长度分别表示为 CODE_XXX_BASE 和 CODE_XXX_SIZE，数据段的长度和基址分别表示为 DATA_XXX_BASE 和 DATA_XXX_SIZE，设备段的长度和基址分别表示为 DEVICE_XXX_BASE 和 DEVICE_XXX_SIZE。

如果用户不填写该字段，那么该内存段将作为匿名内存段存在，上述宏将不会被定义。

5.3.14.14.2 私有段基址<Base>

该配置项是一个可选的十六进制值。用户应该填入该共享内存段的起始地址。如果填写“Auto”，则代表该共享内存段的位置由编译器自动分配。

5.3.14.14.3 私有段长度<Size>

该配置项是一个必填的十六进制值。用户应该填入该共享内存段的长度。

5.3.14.14.4 私有段对齐方式<Align>

该配置项是一个必填的十进制值，决定了该存储器的对齐方式。

当段基址为“Auto”时，如果填写为 N，则该段存储器在自动分配时对齐到 2^N；如果填写为“Auto”，则代表该共享内存段的对齐方式由编译器自主决定。

当段基址是固定值时，该配置项必须填写“Auto”。

对于同一段内存，不同的架构支持的对齐方式可能有不同。

5.3.14.14.5 私有段类别<Type>

该配置项是决定了内存段的类别。可以选择的类别有三种：代码（Code）、数据（Data）、设备（Device）。编译器在分配内存时，会使用这个类别决定将该共享内存段分配到哪一种存储器。

5.3.14.14.6 私有段属性<Attribute>

该配置项决定了内存段的属性；有关属性的内容，请参见 5.3.11.6。

5.3.14.15 共享存储块引用<Shmem><Sn>

本配置组包含了该进程要引用的共享内存块的属性。

5.3.14.15.1 共享段名称<Name>

该配置项决定了要添加进本内存的共享内存段。该共享内存段必须在 5.3.11 中声明。

5.3.14.15.2 共享段属性<Attribute>

该配置项决定了共享存储块映射进本进程时的属性，该属性必须是共享内存段声明时的属性的一个子集；这实现了细粒度的受控共享。比如，某共享存储块声明时的属性为 RWBCS，但在本进程中引用时属性为 RBCS，则本进程只能读取而不能写入该共享存储块[1]。有关属性的内容，请参见 5.3.11.6。

5.3.14.16 线程<Thread><Tn>

本配置组包含了在该进程中运行的线程的属性。每一个进程中至少要有一个线程存在。

此配置组仅适用于原生进程，对虚拟机不存在。

5.3.14.16.1 线程名称<Name>

该配置项是一个必填的字段，决定了线程的名称。该名称必须是一个合法 C 标识符，而且在同一个进程内应当是唯一的。

5.3.14.16.2 线程栈大小<Stack_Size>

该配置项是一个必填的十六进制值，决定了线程栈的大小。该大小可根据需求斟酌决定。

5.3.14.16.3 线程参数<Parameter>

该配置项是一个可选的十六进制值，决定了线程启动时传递给它的参数。如果不填写，那么默认为 0。

5.3.14.16.4 线程优先级<Priority>

该配置项是一个必填的整数值，决定了线程的实时优先级。该优先级必须在 Kern_Prio-2 和 5 之间[2]。比如，如果内核优先级数 Kern_Prio 是 32，那么该值可以在 5 和 30 之间挑选。该值越大，那么线程的优先级越高。

---

[1] 其它引用本共享存储块且属性包含 W 的进程才有写权限

[2] 最高优先级 Kern_Prio-1 被安全守护线程 SFTD 占据，而优先级 4 及以下被 RVM 及其负责管理的虚拟机使用

5.3.14.17 迁移调用目标<Invocation><In>

本配置组包含了在该进程中设置的迁移调用目标的相关信息。在进程中可以有迁移调用目标也可以没有；当其他进程内的线程激活迁移调用目标时，即跳转到该迁移调用目标处执行，执行完后再返回原进程。

此配置项仅适用于原生进程，对虚拟机不存在。

5.3.14.17.1 迁移调用名称<Name>

该配置项是一个必填的字段，决定了迁移调用的名称。该名称必须是一个合法 C 标识符，而且在同一个进程内应当是唯一的。

5.3.14.17.2 迁移调用栈大小<Stack_Size>

该配置项是一个必填的十六进制值，决定了迁移调用栈的大小。该大小可根据需求斟酌决定。

5.3.14.18 迁移调用入口<Port><Pn>

本配置组包含了从该进程可以调用的迁移调用的相关信息。在进程中可以有迁移调用入口也可以没有；迁移调用入口一旦激活，激活它的线程就会跳转到迁移调用目标处执行，并在完成后返回自己的进程继续执行。

此配置项仅适用于原生进程，对虚拟机不存在。

5.3.14.18.1 迁移调用目标名称<Name>

该配置项决定了迁移调用要进入的具体目标。

5.3.14.18.2 迁移调用目标所在进程名称<Process>

该配置项决定了迁移调用目标所在进程。对一个迁移调用的引用由目标名称和目标所在进程名称唯一确定。

5.3.14.19 信号接收端点<Receive><Rn>

该配置组包含了该进程中的线程可以阻塞接收信息的所有接收信号端点。在进程中可以有接收信号端点也可以没有；一旦与接收信号端点对应的发送信号端点遭到激活，接收信号端点上的线程就会解除等待开始运行，直到其再次阻塞。

5.3.14.19.1 接收端点名称<Name>

该配置项是一个必填的字段，决定了接收端点的名称。该名称必须是一个合法 C 标识符，而且在同一个进程内应当是唯一的。在进程的主头文件 prc_xxx.h 中将生成各接收端点的形如 RCV_XXX 的宏定义，以方便用户在 RME_Sig_Rcv 等系统调用中引用。

5.3.14.20 信号发送端点<Send><Sn>

本配置组包含了该进程中的线程可以发送到的所有发送信号端点。在进程刚刚建立时，该项为空。在进程中可以有发送信号端点也可以没有；一旦发送信号端点遭到激活，与之对应的接收信号端点上的线程就会解除等待开始运行，直到其再次阻塞。

此配置项在虚拟机中也可以存在，此时它作为虚拟机向原生进程发送信号的重要手段。更多信息请参见 2.2.7.2。

5.3.14.20.1 信号接收端点名称<Name>

该配置项决定了该信号发送端点对应的信号接收端点的名称。在进程的主头文件 prc_xxx.h 中将生成各发送端点的形如 SND_XXX_PRC_XXX 的宏定义，以方便用户在 RME_Sig_Snd 等系统调用中引用。

5.3.14.20.2 信号接收端点所在进程名称<Process>

该配置项决定了信号接收端点所在进程。对一个信号接收端点的引用由端点名称和端点所在进程名称唯一确定。

5.3.14.21 向量接收端点<Vector><Vn>

本配置组包含了该进程中的线程可以阻塞接收的所有物理中断的向量信号端点。在进程中可以有物理向量接收端点也可以没有；一旦该中断向量遭到激活，与之对应的物理向量端点上的线程就会解除等待开始运行，直到其再次阻塞。

此配置项仅适用于原生进程，对虚拟机不存在。

5.3.14.21.1 向量端点名称<Name>

该配置项决定了物理中断向量的名称。该名称可由用户自行决定。在进程的主头文件 prc_xxx.h 中将生成各发送端点的形如 VCT_XXX 的宏定义，以方便用户在 RME_Sig_Rcv 等系统调用中引用。

5.3.14.21.2 向量号<Number>

该配置项决定了物理中断向量的向量号。该向量号必须小于芯片描述文件中声明的向量总数，且每一个向量号只能在配置文件中出现一次。

5.3.14.22 内核功能调用<Kfunc><Kn>

本配置组包含了该进程中的线程可以激活的所有内核功能调用。在进程中可以有内核功能调用也可以没有；进程中的线程可以通过激活内核调用来访问一些特殊的需要在内核态执行的操作。

此配置项在虚拟机中也可以存在，此时它可用来授权特定的虚拟机进行一些内核级敏感操作。需要注意的是，如果内核功能调用被滥用，系统的安全性会遭到严重损害，因此任何内核功能授权行为都需要进行全面的系统安全评估后慎重做出。

5.3.14.22.1 功能调用名称<Name>

该配置项决定了该内核功能调用的名称。该名称必须是一个合法 C 标识符，而且在同一个进程内应当是唯一的。在进程的主头文件 prc_xxx.h 中将生成各发送端点的形如 KFN_XXX 的宏定义，以方便用户在 RME_Kfn_Act 等系统调用中引用。

5.3.14.22.2 主功能号起始值<Begin>

该配置项是一个十六进制值，决定了该内核功能调用允许的主功能号的开始点。

5.3.14.22.3 主功能号结束值<End>

该配置项是一个十六进制值，决定了该内核功能调用允许的主功能号的结束点。这个值不能比起始值低。如一个内核功能调用的主功能号开始点为 0x10，结束点为 0x1F，那么可以调用的内核功能的主功能号在 0x10 到 0x1F 之间，包括了 0x10 和 0x1F。

5.3.15 虚拟机配置<Virtual><Vn>

本配置组负责系统中的虚拟机的配置。系统中可以有一个或多个虚拟机，因此该类可以有一个或多个实例。虚拟机实际上是基于原生进程实现的，它拥有原生进程的大多数配置项，但又有所不同。相比原生进程，虚拟机不能拥有线程<Thread>、迁移调用目标<Invocation>、迁移调用入口<Port>、信号接收端点<Receive>以及向量接收端点<Vector>，但却可以拥有私有存储块<Memory>、共享存储块引用<Shmem>以及信号发送端点<Send>。此外，相比原生进程，虚拟机还额外多出一些配置项；接下来将对这些虚拟机特有的配置项进行一一讲解。

5.3.15.1 中断处理线程栈大小<Vector_Stack_Size>

该配置项是一个十六进制值，决定了虚拟机中断处理线程 Vct 的栈大小。该大小可以比照裸机 RTOS 中的中断向量专用堆栈的大小，具体数值需要视乎登记的虚拟中断向量处理函数使用的栈深度。一般地，512B 是一个很好的参考值。

5.3.15.2 用户程序线程栈大小<User_Stack_Size>

该配置项是一个十六进制值，决定了虚拟机用户程序线程 Usr 的栈大小。该大小可以比照裸机 RTOS 中的启动时堆栈大小，它一般不需要很大，只要能够完成 RTOS 自身的初始化即可；RTOS 启动后一般不会再被使用该栈，而是会为每个 RTOS 线程分配专属的栈。一般地，256B 是一个很好的参考值。

5.3.15.3 优先级<Priority>

该配置项是一个整数值，决定了虚拟机的虚拟化优先级。该优先级必须在虚拟机优先级数-1 和 0 之间。比如，如果虚拟化优先级数是 32，那么该值可以在 0 和 31 之间挑选。该值越大，那么虚拟机的优先级越高。

需要注意的是，虚拟机优先级和原生进程中的线程优先级不是一回事；虚拟机的优先级比任何原生进程都低，它使用的优先级是虚拟化优先级。只有当所有原生进程中的线程都阻塞时，高优先级虚拟机才运行；当高优先级的虚拟机完全阻塞不运行时，才轮到低优先级虚拟机运行。

5.3.15.4 时间片<Timeslice>

该配置项是一个大于 0 的整数值，决定了虚拟机的时间片数量，单位为 RVM 时钟嘀嗒[1]数。在系统中，不同优先级的虚拟机之间为全抢占关系，而相同优先级的虚拟机之间则为时间片轮转调度关系。一般推荐将这个值设的比较大，比如 50 或 100，这样可以避免频繁进行虚拟机间的切换，提高系统的性能。

5.3.15.5 时钟周期<Period>

该配置项是一个大于 0 的整数值，决定了虚拟机的虚拟时钟中断的分频系数，单位为 RVM 时钟嘀嗒数：每经过 Period 个 RVM 时钟嘀嗒，才会向本虚拟机发送一次定时器中断 Tim[2]。一般推荐将这个值设的比较大，比如 5 或 10，以防止时钟嘀嗒大量消耗系统资源。

5.3.15.6 看门狗超时时间<Watchdog>

该配置项是一个整数值，决定了虚拟机的看门狗超时时间，单位为 RVM 时钟嘀嗒数。一旦某虚拟机失去响应而导致看门狗超时，RVM 即会重启该超时的虚拟机。通常而言看门狗超时时间推荐设为 1 秒左右，这大概是 100 个或 1000 个 RVM 时钟嘀嗒。如果本配置项设置为 0，那么意味着本虚拟机不启用看门狗。

看门狗超时时间是按照虚拟机运行时间来计算的，而非真实系统的运行时间。假设看门狗超时时间设置为 1 秒，那么意味着该虚拟机的净运行时间需要超过 1 秒且未调用 RVM_Hyp_Wdg_Clr 喂狗才会导致看门狗超时。

5.3.15.7 虚拟中断向量数<Vect_Num>

该配置项决定了虚拟机的虚拟中断源的数量。推荐在够用的前提下将该值设置得尽量小，以节约系统资源。

5.3.15.8 客户机操作系统类型<Guest_Type>

该配置项决定了运行的客户机操作系统的类型。

5.3.15.9 客户机操作系统根目录<Guest_Root>

该配置项决定了客户机操作系统的根目录，是一个相对于工程描述文件的路径。生成的虚拟机工程会从这个目录引用客户机操作系统的文件。

---

[1] 请参见 2.1.2.4，下同

[2] 请参见 2.2.4.1

5.3.15.10 客户机头文件输出目录<Virtual_Header_Output>

该配置项决定了客户机 RTOS 头文件的输出目录，是一个相对于工程描述文件的路径。具体生成什么配置头文件，以及其内容如何，由选择的具体客户机操作系统而定。

5.3.15.11 客户机头文件覆盖<Virtual_Header_Overwrite>

决定在重新生成时是否覆盖之前生成的客户机 RTOS 头文件，可以是 Yes 或 No。如果用户手动编辑了客户机头文件而希望保留这些更改，可以选择 No 以保留原来的老版本。值得注意的是，即便选择了 No，具体哪些头文件被保留、哪些头文件被覆盖仍是由具体的客户机生成工具决定的，但生成工具会尽量使那些用户可编辑的头文件不被覆盖。

5.3.15.12 客户机头文件输出目录<Virtual_Source_Output>

该配置项决定了客户机 RTOS 源文件的输出目录，是一个相对于工程描述文件的路径。具体生成什么配置源文件，以及其内容如何，由选择的具体客户机操作系统而定。

5.3.15.13 客户机头文件覆盖<Virtual_Header_Overwrite>

决定在重新生成时是否覆盖之前生成的客户机 RTOS 源文件，可以是 Yes 或 No。如果用户手动编辑了客户机源 RTOS 文件而希望保留这些更改，可以选择 No 以保留原来的老版本。值得注意的是，即便选择了 No，具体哪些源文件被保留、哪些源文件被覆盖仍是由具体的客户机生成工具决定的，但生成工具会尽量使那些用户可编辑的源文件不被覆盖。

5.4 平台描述文件结构<Platform>

本文件负责记载平台的基本信息。工程生成器将用这些信息来判断平台的具体型号和兼容性。

5.4.1 名称<Name>

架构的名称。必须和平台的标准名称[1]一致。工程生成器将依赖该名称来查找平台描述文件，并交叉确认工程文件、平台描述文件和芯片描述文件中的架构缩写一致。

5.4.2 版本<Version>

该配置项记录了平台描述文件的版本。

工程生成器不关心该配置项的值。

5.4.3 字长<Wordlength>

[1] 平台的文件夹缩写名

该配置项记录该平台的处理器字长，单位为位。典型值为 32 或 64。

5.4.4 允许协处理器<Coprocessor>

该配置项是一个逗号分隔列表，它记录了架构中允许的协处理器，如 FPU 的型号等。工程生成器将据此检查具体的芯片配置文件的合法性。如果某种架构没有协处理器，那么可以空着不填。

5.4.5 客户机列表<Guest>

该配置项是一个逗号分隔列表，记录了所有该架构允许的客户机操作系统。

5.4.6 构建系统列表<Buildsystem>

该配置项是一个逗号分隔列表，记录了所有该架构允许的构建系统。

5.4.7 工具链列表<Toolchain>

该配置项是一个逗号分隔列表，记录了所有该架构允许的编译工具链。

5.4.8 架构选项<Config><Cn>

本配置组负责记载架构本身具备的固有选项。工程编译器会试图从工程中查询这些固有选项的配置。

5.4.8.1 名称<Name>

该配置项决定了该选项的内部名称。工程生成器将使用这个名称到工程文件中检索对应的选项。

5.4.8.2 类别<Type>

该配置项是一个字段，决定了该选项的性质。类别只有两种，一种是 Choice（选择），另外一种是 Range（范围）。前者代表可以在<Range>标签列出的各个值之中选择一个作为其值，后者代表<Range>标签列出的两个值之间的所有值都可以。

5.4.8.3 宏定义<Macro>

该配置项是一个字段，决定了该选项对应的宏名。工程生成器会根据该宏名生成配置文件。

5.4.8.4 取值范围<Range>

该配置项是一个逗号分隔符字段。当选项是 Choice 类型时，该分隔符字段列出所有选项可能取的值，这些值可以是数字也可以是任意字符串；当选项是 Range 类型，该分隔符字段中只能有两个非负整型数值，这两个值决定了取值范围。

5.5 芯片描述文件结构<Chip>

本文件负责记载芯片的基本信息。这些基本信息将被生成器用来判断芯片的具体型号和兼容性。

5.5.1 名称<Name>

该配置项是一个字段，决定了芯片大类名称。凡属于这个类别的芯片都使用这个芯片 XML 进行描述。通常而言，会放在一个类别内的各个芯片仅仅具有管脚数量、外设数量的细微差异，其存储器布局都应当完全一致；芯片描述文件的后续配置项会详细定义它们。

5.5.2 版本<Version>

该配置项记录了芯片描述文件的版本。

工程生成器不关心该配置项的值。

5.5.3 平台架构<Platform>

该配置项决定了芯片的凭他架构。该名称必须与芯片位于的平台文件夹的简称一致。工程生成器将交叉确认工程文件、平台描述文件和芯片描述文件中的架构缩写一致。

5.5.4 兼容列表<Compatible>

该配置项是一个逗号分隔列表，决定了本描述文件兼容的具体的芯片的全名。只有列出在该字段内的芯片才可以被兼容；工程描述文件中给出的芯片具体型号必须在此列表中。

5.5.5 厂商<Vendor>

该配置项决定了芯片的制造厂商。它在生成某些工具链的工程时会用到。

5.5.6 中断向量数<Vector>

该配置项决定了芯片的外部设备中断向量的数量。

5.5.7 通用内存保护区域数<Region>

该配置项决定了芯片的通用 MPU 区域数量。通用 MPU 区域既可以用于代码段，也可以用于数据段和设备段。

5.5.8 代码内存保护区域数<Iregion>

该配置项决定了芯片的代码专用 MPU 区域数量。这些 MPU 区域与通用 MPU 区域不同，它们只能用于代码段。

工程生成器暂时不读取该选项。

5.5.9 数据内存保护区域数<Dregion>

该配置项是一个十进制数，决定了芯片的数据专用 MPU 区域数量。这些 MPU 区域与通用 MPU 区域不同，它们只能用于数据段和设备段。

工程生成器暂时不读取该选项。

5.5.10 存在协处理器<Coprocessor>

该配置项是一个逗号分隔列表，它记录了具体芯片中存在的协处理器，如 FPU 的型号等。工程生成器将根据架构描述文件检测它的合法性，也将据此检查具体的工程描述文件的合法性。如果某个具体芯片没有协处理器，那么可以空着不填。

5.5.11 架构属性组<Attribute>

本配置组决定了架构相关的特有属性。这些信息随着每个架构都有可能不同，如 CPU 的具体类型、FPU 的有无及型号，等等。这些属性不能由用户设置，是固有于芯片本身的配置项。

5.5.12 芯片固有存储器<Memory><Mn>

本配置组负责记载芯片的片上存储器信息。芯片上存在的一切片上存储器都必须被登录在此。具体的登录方法与额外存储器[1]是一样的。

5.5.12.1 存储段基址<Base>

该配置项是一个十六进制值，决定了该内存段的起始地址。

5.5.12.2 存储段长度<Size>

该配置项是一个十六进制值，决定了该内存段的长度。

5.5.12.3 存储段类别<Type>

该配置项是一个字段，决定了内存段的类别。可能的类别有三种：代码（Code）、数据（Data）和设备（Device）。

5.5.12.4 存储段属性<Attribute>

该配置项是一个字段，决定了内存段的属性。内存段属性分别有可读（Readable，R）、可写（Writable，W）、可执行（eXecutable，X）、可缓冲（Bufferable，B）、可缓存（Cacheable，C）、静态映射（Static，S）六种，它们可以以任意组合进行叠加。

一般地，代码段 ROM 的属性应设置为 RXBCS，数据段 RAM 的属性应设置为 RWBCS。设备段的属性则应设置为 RWS。

5.5.13 芯片选项<Config><Cn>

[1] 请参见 5.3.9

本类负责记载芯片本身具备的固有选项[1]。工程编译器会试图从工程中查询这些固有选项的配置。这些选项的组织方式与架构固有选项是一样的。

5.5.13.1 名称<Name>

该配置项决定了该选项的内部名称。工程生成器将使用这个名称到工程文件中检索对应的选项。

5.5.13.2 类别<Type>

该配置项是一个字段，决定了该选项的性质。类别只有两种，一种是 Choice（选择），另外一种是 Range（范围）。前者代表可以在<Range>标签列出的各个值之中选择一个作为其值，后者代表<Range>标签列出的两个值之间的所有值都可以。

5.5.13.3 宏定义<Macro>

该配置项是一个字段，决定了该选项对应的宏名。工程生成器会根据该宏名生成配置文件。

5.5.13.4 取值范围<Range>

该配置项是一个逗号分隔符字段。当选项是 Choice 类型时，该分隔符字段列出所有选项可能取的值，这些值可以是数字也可以是任意字符串；当选项是 Range 类型，该分隔符字段中只能有两个非负整型数值，这两个值决定了取值范围。

5.6 工程生成器错误信息

5.6.1 命令行参数分析错误（C0000-C9999）

本工具支持的所有输出格式如下：

表 5-1 支持的输出工具链

输出工具链选项	工程组织	编译器
makefile	Makefile	GCC
keil	Keil uVision	ARMCC 5.0

本步骤可能产生的所有报错信息如下（C0000 -- C9999）：

表 5-2 命令行参数分析可能产生的报错信息

错误编号	错误信息	原因
C0000	Too many or too few input parameters.	输入了过少或过多的命令行参数。

[1] 不包括架构的固有选项；那些选型在平台描述文件中列出，而非这里；请参见 5.4.7

错误编号	错误信息	原因
C0001	More than one input file.	指定了多于一个的输入文件。
C0002	More than one output path.	指定了多于一个的输出路径。
C0003	More than one output format.	指定了多于一个的输出格式。
C0004	Unrecognized command line argument.	未能识别的命令行参数。
C0005	No input file specified.	没有指定输入文件。
C0006	No output path specified.	没有指定输出路径。
C0007	No output project format specified.	没有指定输出工程格式。

5.6.2 工程 XML 文件分析错误（P0000-P9999）

该模块分析工程描述 XML 文件，也即 GUI 模块生成的 XML 文件。它首先按照输入的路径读取 XML 文件生成其 DOM，然后逐步分析该 DOM 中的各个组件，并最终生成方便引用的内部数据结构。下面将分章节讲解每个分步骤的具体内容。

5.6.2.1 工程配置分析（P0000-P0099）

在本步骤中，系统中的额外内存信息、共享内存信息和一些工程基本信息会经分析后直接填充在内部数据结构中。可能产生的所有报错信息如下（P0000 -- P0099）：

表 5-3 工程配置初步分析阶段可能产生的报错信息

错误编号	错误信息	原因
P0000	Project XML parsing failed.	工程 XML 文件存在语法错误，无法解析。
P0001	Project XML is malformed.	工程 XML 中不存在<Project>标签。
P0002	Name section is missing.	不存在。
P0003	Name section is empty.	为空。
P0004	Platform section is missing.	目标平台<Platform>不存在。
P0005	Platform section is empty.	目标平台<Platform>为空。
P0006	Chip class section is missing.	芯片名称<Name>不存在。
P0007	Chip class section is empty.	芯片名称<Name>为空。
P0008	Chip fullname section is missing.	芯片类别<Class>不存在。
P0009	Chip fullname section is empty.	芯片类别<Class>为空。
P0010	RME section is missing.	工程 XML 中不存在<RME>标签。
P0011	RVM section is missing.	工程 XML 中不存在<RVM>标签。

错误编号	错误信息	原因
P0012	Extra memory section is missing.	工程 XML 中不存在<Extmem>标签。
P0013	Extra memory section parsing internal error.	解析<Extmem>标签及其内容时遇到未知错误。请检查该标签的完整性。
P0014	Extra memory type section is missing.	额外段类别<Type>不存在。
P0015	Extra memory type section is empty.	额外段类别<Type>为空。
P0016	Extra memory type is malformed.	额外段类别<Type>必须是 Code，Data 或 Device 三者之一。当前值无法识别。
P0017	Shared memory section is missing.	工程 XML 中不存在<Shmem>标签。
P0018	Shared memory section parsing internal error.	解析<Shmem>标签及其内容时遇到未知错误。请检查该标签的完整性。
P0019	Shared memory type section is missing.	共享段类别<Type>不存在。
P0020	Shared memory type section is empty.	共享段类别<Type>为空。
P0021	Shared memory name cannot be omitted.	共享段名称<Name>是必填的，它不可以被省略不写。
P0022	Shared memory type is malformed.	额外段类别<Type>必须是 Code，Data 或 Device 三者之一。当前值无法识别。
P0023	Process section parsing internal error.	解析<Process>标签及其内容时遇到未知错误。请检查该标签的完整性。
P0024	Virtual machine section parsing internal error.	解析<Virtual>标签及其内容时遇到未知错误。请检查该标签的完整性。
P0025	Project XML and chip XML platform does not match.	工程 XML 和芯片 XML 中的平台类型不对应。请检查工程 XML 是否指定了错误的平台。
P0026	Project XML and chip XML chip class does not match.	工程 XML 和芯片 XML 中的芯片类别不对应。请检查工程 XML 是否指定了错误的芯片类别。
P0027	The specific chip designated in project XML not found in chip XML.	工程 XML 中指定的芯片在芯片 XML 中不被兼容。请检查是否指定了错误的芯片型号。
P0028	The specific platform is currently not supported.	工程 XML 中指定的平台现在不被支持。请检查是否指定了错误的平台。

5.6.2.2 内核配置分析（P0100-P0199）

本阶段分析内核的基本配置信息，它们都位于<RME>标签下。可能产生的所有报错如下表所示（P0100 -- P0199）：

表 5-4 内核配置分析阶段可能产生的报错信息

错误编号	错误信息	原因
P0100	Code base section is missing.	代码段基址<Code_Base>不存在。
P0101	Code base is not a valid hex integer.	代码段基址<Code_Base>不是十六进制。
P0102	Code size section is missing.	代码段长度<Code_Size>不存在。
P0103	Code size is not a valid hex integer.	代码段长度<Code_Size>不是十六进制。
P0104	Data base section is missing.	数据段基址<Data_Base>不存在。
P0105	Data base is not a valid hex integer.	数据段基址<Data_Base>不是十六进制。
P0106	Data size section is missing.	数据段长度<Data_Size>不是十六进制。
P0107	Data size is not a valid hex integer.	数据段长度<Data_Size>不是十六进制。
P0108	Stack size section is missing.	内核栈长度<Stack_Size>不存在。
P0109	Stack size is not a valid hex integer.	内核栈长度<Stack_Size>不是十六进制。
P0110	Extra kernel memory section is missing.	额外内核对象用内存长度<Extra_Kom>不存在。
P0111	Extra kernel memory is not a valid hex integer.	额外内核对象用内存长度<Extra_Kom>不是十六进制。
P0112	Kernel memory order section is missing.	内核内存分配粒度<Kom_Order>不存在。
P0113	Kernel memory order is not a valid unsigned integer.	内核内存分配粒度<Kom_Order>不是十进制。
P0114	Priority number section is missing.	优先级数量<Kern_Prio>不存在。
P0115	Priority number is not a valid unsigned integer.	优先级数量<Kern_Prio>不是十进制。
P0116	Compiler option section is missing.	<Compiler>标签不存在。
P0117	Platform option section is missing.	<Platform>标签不存在。
P0118	Platform option section parsing internal error.	解析<Platform>标签及其内容时遇到未知错误。请检查该标签的完整性。
P0119	Chip option section is missing.	<Chip>标签不存在。
P0120	Chip option section parsing internal	解析<Chip>标签及其内容时遇到未知错误。请

错误编号	错误信息	原因
	error.	检查该标签的完整性。
P0121	Chip option mismatch in project option list.	<Chip>标签下所提供的某个选项在芯片 XML 中找不到，或者并非芯片 XML 中的所有选项都被指定。请检查是否指定了错误或多余的标签，或者漏掉了某些标签。
P0122	Value conversion failure.	某 Range 类型选项的设置值转换失败了。请检查工程 XML 中该选项是否被设置为无符号整数。
P0123	Value overflow.	某 Range 类型选项的设置值超出了允许范围的上界。请检查并重新设置。
P0124	Value underflow.	某 Range 类型选项的设置值超出了允许范围的下界。请检查并重新设置。
P0125	Value not found.	某 Select 类型选项的设置值不在允许的选项列表中。请检查并重新设置。

5.6.2.3 用户态库配置分析（P0200-P0299）

本阶段分析用户态库的基本配置信息，它们都位于<RVM>标签下。可能产生的所有报错如下表所示（P0200 -- P0299）：

表 5-5 用户态库配置分析阶段可能产生的报错信息

错误编号	错误信息	原因
P0200	Code size section is missing.	代码段长度<Code_Size>不存在。
P0201	Code size is not a valid hex integer.	代码段长度<Code_Size>不是十六进制。
P0202	Data size section is missing.	数据段长度<Data_Size>不存在。
P0203	Data size is not a valid hex integer.	数据段长度<Data_Size>不是十六进制。
P0204	Stack size section is missing.	启动守护线程栈长度<Init_Stack_Size>不存在。
P0205	Stack size is not a valid hex integer.	启动守护线程栈长度<Init_Stack_Size>不是十六进制。
P0206	Extra capability table size section is missing.	额外权能表槽位数<Extra_Captbl>不存在。
P0207	Extra capability table size is not a valid unsigned integer.	额外权能表槽位数<Extra_Captbl>不是十进制。

错误编号	错误信息	原因
P0208	Compiler section is missing.	<Compiler>标签不存在。
P0209	Virtual machine priorities section is missing.	虚拟化优先级数<Virt_Prio>不存在。
P0210	Virtual machine priorities is not a valid unsigned integer.	虚拟化优先级数<Virt_Prio>不是十进制。
P0211	Virtual machine event number section is missing.	虚拟机事件源数<Virt_Event>不存在。
P0212	Virtual machine event number is not a valid unsigned integer.	虚拟机事件源数<Virt_Event>不是十进制。
P0213	Virtual machine mapping total number section is missing.	虚拟机向量映射数<Virt_Map>不存在。
P0214	Virtual machine mapping total number is not a valid unsigned integer.	虚拟机向量映射数<Virt_Map>不是十进制。

5.6.2.4 进程配置分析（P0300-P0399）

本阶段分析各个进程的基本配置信息，它们都位于<Process>标签下。可能产生的所有报错如下表所示（P0300 -- P0399）：

表 5-6 进程配置分析阶段可能产生的报错信息

错误编号	错误信息	原因
P0300	Name section is missing.	进程名称<Name>不存在。
P0301	Name section is empty.	进程名称<Name>为空。
P0302	Extra capability table size section is missing.	额外权能表槽位数<Extra_Captbl>不存在。
P0303	Extra capability table size is not a valid unsigned integer.	额外权能表槽位数<Extra_Captbl>不是十进制。
P0304	Compiler section is missing.	<Compiler>标签不存在。
P0305	Memory section is missing.	<Memory>标签不存在。
P0306	Memory section parsing internal error.	解析<Memory>标签及其内容时遇到未知错误。请检查该标签的完整性。
P0307	Memory section is empty.	<Memory>标签为空，这意味着无私有内存段。
P0308	Memory type section is missing.	**错误！未定义书签。**不存在。
P0309	Memory type section is empty.	**错误！未定义书签。**为空。

错误编号	错误信息	原因
P0310	Memory type is malformed.	**错误！未定义书签。**必须是 Code，Data 或 Device 三者之一。当前值无法识别。
P0311	Shared memory section is missing.	<Shmem>标签不存在。
P0312	Shared memory section parsing internal error.	解析<Shmem>标签及其内容时遇到未知错误。请检查该标签的完整性。
P0313	Shared memory type section is missing.	**错误！未定义书签。**不存在。
P0314	Shared memory type section is empty.	**错误！未定义书签。**为空。
P0315	Shared memory type is malformed.	**错误！未定义书签。**必须是 Code，Data 或 Device 三者之一。当前值无法识别。
P0316	Thread section is missing.	<Thread>标签不存在。
P0317	Thread section parsing internal error.	解析<Thread>标签及其内容时遇到未知错误。请检查该标签的完整性。
P0318	Thread section is empty.	<Thread>标签为空，这意味着无线程存在。
P0319	Invocation section is missing.	<Invocation>标签不存在。
P0320	Invocation section parsing internal error.	解析<Invocation>标签及其内容时遇到未知错误。请检查该标签的完整性。
P0321	Port section is missing.	<Port>标签不存在。
P0322	Port section parsing internal error.	解析<Port>标签及其内容时遇到未知错误。请检查该标签的完整性。
P0323	Receive section is missing.	<Receive>标签不存在。
P0324	Receive section parsing internal error.	解析<Receive>标签及其内容时遇到未知错误。请检查该标签的完整性。
P0325	Send section is missing.	<Send>标签不存在。
P0326	Send section parsing internal error.	解析<Send>标签及其内容时遇到未知错误。请检查该标签的完整性。
P0327	Vector section missing.	<Vector>标签不存在。
P0328	Vector section parsing internal error.	解析<Vector>标签及其内容时遇到未知错误。请检查该标签的完整性。
P0329	Kernel function section missing.	<Kernel>标签不存在。
P0330	Kernel function section parsing internal error.	解析<Kernel>标签及其内容时遇到未知错误。请检查该标签的完整性。
P0331	No primary code section exists.	该进程中必须至少有一个私有的 Code 内存段

错误编号	错误信息	原因
		作为主代码内存段，用来存放代码。
P0332	No primary data section exists.	该进程中必须至少有一个私有的 Data 内存段作为主数据内存段，用来存放数据。
P0333	Primary code section does not have RXS attribute.	进程的主代码段没有具备完整的 RXS 属性（可读可执行，且静态映射）。它必须具备该属性才能作为主代码段。
P0334	Primary data section does not have RWS attribute.	进程的主数据段没有具备完整的 RWS 属性（可读写且静态映射）。它必须具备该属性才能作为主数据段。
P0335	No thread exists.	进程中至少要有一个线程，现在没有。

5.6.2.5 虚拟机配置分析（P0400-P0499）

本阶段分析各个虚拟机的基本配置信息，它们都位于<Virtual>标签下。可能产生的所有报错如下表所示（P0400 -- P0499）：

表 5-7 虚拟机配置分析阶段可能产生的报错信息

错误编号	错误信息	原因
P0400	Name section is missing.	**错误！未定义书签。**不存在。
P0401	Name section is empty.	**错误！未定义书签。**为空。
P0402	Guest operating system selection section is missing.	**错误！未定义书签。**不存在。
P0403	Guest operating system selection section is empty.	**错误！未定义书签。**为空。
P0404	Extra capability table size section is missing.	**错误！未定义书签。**不存在。
P0405	Extra capability table size is not a valid unsigned integer.	**错误！未定义书签。**不是十进制。
P0406	Stack size section is missing.	**错误！未定义书签。**不存在。
P0407	Stack size is not a valid hex integer.	**错误！未定义书签。**不是十六进制。
P0408	Priority section is missing.	**错误！未定义书签。**不存在。
P0409	Priority is not a valid unsigned integer.	**错误！未定义书签。**不是十六进制。
P0410	Timeslices section is missing.	**错误！未定义书签。**不存在。
P0411	Timeslices is not a valid unsigned	**错误！未定义书签。**不是十进制。

错误编号	错误信息	原因
	integer.	
P0412	Timeslices cannot be zero.	**错误！未定义书签。**不能为 0。
P0413	Timer interrupt period section is missing.	**错误！未定义书签。**不存在。
P0414	Timer interrupt period is not a valid unsigned integer.	**错误！未定义书签。**不是十进制。
P0415	Timer interrupt period cannot be zero.	**错误！未定义书签。**不能为 0。
P0416	Watchdog timeout section is missing.	**错误！未定义书签。**不存在。
P0417	Watchdog timeout is not a valid unsigned integer.	**错误！未定义书签。**不是十进制。
P0418	Virtual vector number section is missing.	**错误！未定义书签。**不存在。
P0419	Virtual vector number is not a valid unsigned integer.	**错误！未定义书签。**不是十进制。
P0420	Virtual vector number too large.	**错误！未定义书签。**大于 768，不支持。
P0421	Virtual vector number cannot be zero.	**错误！未定义书签。**不能为 0。
P0422	Compiler section is missing.	<Compiler>标签不存在。
P0423	Memory section is missing.	<Memory>标签不存在。
P0424	Memory section parsing internal error.	解析<Memory>标签及其内容时遇到未知错误。请检查该标签的完整性。
P0425	Memory section is empty.	<Memory>标签为空，这意味着无私有内存段。
P0426	Memory type section is missing.	**错误！未定义书签。**不存在。
P0427	Memory type section is empty.	**错误！未定义书签。**为空。
P0428	Memory type is malformed.	**错误！未定义书签。**必须是 Code，Data 或 Device 三者之一。当前值无法识别。
P0429	Shared memory section is missing.	<Shmem>标签不存在。
P0430	Shared memory section parsing internal error.	解析<Shmem>标签及其内容时遇到未知错误。请检查该标签的完整性。
P0431	Shared memory type section is missing.	**错误！未定义书签。**不存在。
P0432	Shared memory type section is empty.	**错误！未定义书签。**为空。
P0433	Shared memory type is malformed.	**错误！未定义书签。**必须是 Code，Data 或 Device 三者之一。当前值无法识别。

错误编号	错误信息	原因
P0434	Send section is missing.	<Send>标签不存在。
P0435	Send section parsing internal error.	解析<Send>标签及其内容时遇到未知错误。请检查该标签的完整性。
P0436	Kernel function section missing.	<Kernel>标签不存在。
P0437	Kernel function section parsing internal error.	解析<Kernel>标签及其内容时遇到未知错误。请检查该标签的完整性。
P0438	No primary code section exists.	该虚拟机中必须至少有一个私有的 Code 内存段作为主代码内存段，用来存放代码。
P0439	No primary data section exists.	该虚拟机中必须至少有一个私有的 Data 内存段作为主数据内存段，用来存放数据。
P0440	Primary code section does not have RXS attribute.	虚拟机的主代码段没有具备完整的 RXS 属性（可读可执行，且静态映射）。它必须具备该属性才能作为主代码段。
P0441	Primary data section does not have RWS attribute.	虚拟机的主数据段没有具备完整的 RWS 属性（可读写且静态映射）。它必须具备该属性才能作为主数据段。

5.6.2.6 内存段配置分析（P0500-P0599）

内存段配置分析会在分析额外内存（<Extmem>）、系统共享内存(<Project><Shmem>)和各个进程（<Process>）或虚拟机（<Virtual>）的内存（<Memory>）时被调用。它也会在分析芯片自带内存时被调用，因此这些错误也适用于芯片自带内存（<Chip><Memory>，见后文）的分析。可能产生的所有报错如下表所示（P0500 -- P0599）：

表 5-8 内存段配置分析阶段可能产生的报错信息

错误编号	错误信息	原因
P0500	Base section is missing.	额外段基址<Base>、共享段基址<Base>**错误！未定义书签。**、**错误！未定义书签。**或存储段基址<Base>不存在。
P0501	Base is neither automatically allocated nor a valid hex integer.	额外段基址<Base>、共享段基址<Base>**错误！未定义书签。**、**错误！未定义书签。**或存储段基址<Base>不是十六进制，也没有注明自动分配。
P0502	Size section is missing.	额外段长度<Size>、共享段长度<Size>、**错误！未定义书签。**、**错误！未定义书签。**或存储段

错误编号	错误信息	原因
		长度<Size>不存在。
P0503	Size is not a valid hex integer.	额外段长度<Size>、共享段长度<Size>、**错误！未定义书签。**、**错误！未定义书签。**或存储段长度<Size>不是十六进制。
P0504	Size cannot be zero.	额外段长度<Size>、共享段长度<Size>、**错误！未定义书签。**、**错误！未定义书签。**或存储段长度<Size>不能为 0。
P0505	Size is out of bound.	额外段长度<Size>、共享段长度<Size>、**错误！未定义书签。**、**错误！未定义书签。**或存储段长度<Size>的大小超过了 4GB。
P0506	Type section is missing.	额外段类别<Type>、共享段类别<Type>、**错误！未定义书签。**、**错误！未定义书签。**或存储段类别<Type>不存在。
P0507	Device-type memory cannot be automatically allocated.	设备（Device）类别的内存不能被设置为自动分配，必须手动指明其起始地址。
P0508	Attribute section is missing.	额外段属性<Attribute>、共享段属性<Attribute>、**错误！未定义书签。**、**错误！未定义书签。**或存储段属性<Attribute>不存在。
P0509	Attribute section is empty.	额外段属性<Attribute>、共享段属性<Attribute>、**错误！未定义书签。**、**错误！未定义书签。**或存储段属性<Attribute>为空。
P0510	Attribute does not allow any access and is malformed.	额外段属性<Attribute>、共享段属性<Attribute>、**错误！未定义书签。**、**错误！未定义书签。**或存储段属性<Attribute>不允许任何形式的访问（不可读、不可写且不可执行）。

5.6.2.7 共享内存段引用配置分析（P0600-P0699）

本节介绍的共享内存段引用指位于各个进程（<Process>）或虚拟机（<Virtual>）内的对之前声明的共享内存（<Shmem>）的引用，而非共享内存段的声明本身。可能产生的所有报错如下表所示（P0600 -- P0699）：

表 5-9 共享内存段配置分析阶段可能产生的报错信息

错误编号	错误信息	原因
P0600	Name section is missing.	共享段名称<Name>或**错误！未定义书签。**不

错误编号	错误信息	原因
		存在。
P0601	Name section is empty.	共享段名称<Name>或**错误！未定义书签。**为空。
P0602	Attribute section is missing.	共享段属性<Attribute>或**错误！未定义书签。**不存在。
P0603	Attribute section is empty.	共享段属性<Attribute>或**错误！未定义书签。**为空。
P0604	Attribute does not allow any access and is malformed.	共享段属性<Attribute>或**错误！未定义书签。**不允许任何形式的访问（不可读、不可写且不可执行），这是不允许的。

5.6.2.8 线程配置分析（P0700-P0799）

线程配置分析产生会在分析各个进程（<Process>）的线程（<Thread>）时使用。可能产生的所有报错如下表所示（P0700 -- P0799）：

表 5-10 线程配置分析阶段可能产生的报错信息

错误编号	错误信息	原因
P0700	Name section is missing.	线程名称<Name>不存在。
P0701	Name section is empty.	线程名称<Name>为空。
P0702	Stack size section is missing.	线程栈大小<Stack_Size>不存在。
P0703	Stack size is not a valid hex integer.	线程栈大小<Stack_Size>不是十六进制。
P0704	Stack size cannot be zero.	线程栈大小<Stack_Size>不能为 0。
P0705	Parameter section is missing.	线程参数<Parameter>不存在。
P0706	Parameter is not a valid hex integer.	线程参数<Parameter>不是十六进制。
P0707	Priority section is missing.	线程优先级<Priority>不存在。
P0708	Priority is not a valid unsigned integer.	线程优先级<Priority>不是十进制。

5.6.2.9 迁移调用目标配置分析（P0800-P0899）

迁移调用目标配置分析产生会在分析各个进程（<Process>）的迁移调用目标（<Invocation>）时使用。可能产生的所有报错如下表所示（P0800 -- P0899）：

表 5-11 迁移调用目标配置分析阶段可能产生的报错信息

错误编号	错误信息	原因

错误编号	错误信息	原因
P0800	Name section is missing.	迁移调用名称<Name>不存在。
P0801	Name section is empty.	迁移调用名称<Name>为空。
P0802	Stack size section is missing.	迁移调用栈大小<Stack_Size>不存在。
P0803	Stack size is not a valid hex integer.	迁移调用栈大小<Stack_Size>不是十六进制。
P0804	Stack size cannot be zero.	迁移调用栈大小<Stack_Size>不能为 0。

5.6.2.10 迁移调用入口配置分析（P0900-P0999）

迁移调用入口配置分析产生会在分析各个进程（<Process>）的迁移调用入口（<Port>）时使用。可能产生的所有报错如下表所示（P0900 -- P0999）：

表 5-12 迁移调用入口配置分析阶段可能产生的报错信息

错误编号	错误信息	原因
P0900	Name section is missing.	迁移调用目标名称<Name>不存在。
P0901	Name section is empty.	迁移调用目标名称<Name>为空。
P0902	Process name section is missing.	迁移调用目标所在进程名称<Process>不存在。
P0903	Process name section is empty.	迁移调用目标所在进程名称<Process>为空。

5.6.2.11 接收信号端点配置分析（P1000-P1099）

接收信号端点配置分析产生会在分析各个进程（<Process>）的接收信号端点（<Receive>）时使用。可能产生的所有报错如下表所示（P1000 -- P1099）：

表 5-13 接收信号端点配置分析阶段可能产生的报错信息

错误编号	错误信息	原因
P1000	Name section is missing.	接收端点名称<Name>不存在。
P1001	Name section is empty.	接收端点名称<Name>为空。

5.6.2.12 发送信号端点配置分析（P1100-P1199）

发送信号端点配置分析产生会在分析各个进程（<Process>）的发送信号端点（<Receive>）时使用。可能产生的所有报错如下表所示（P1100 -- P1199）：

表 5-14 发送信号端点配置分析阶段可能产生的报错信息

错误编号	错误信息	原因

错误编号	错误信息	原因
P1100	Name section is missing.	信号接收端点名称<Name>或**错误！未定义书签。**不存在。
P1101	Name section is empty.	信号接收端点名称<Name>或**错误！未定义书签。**为空。
P1102	Process name section is missing.	信号接收端点所在进程名称<Process>或**错误！未定义书签。**不存在。
P1103	Process name section is empty.	信号接收端点所在进程名称<Process>或**错误！未定义书签。**为空。

5.6.2.13 物理向量信号端点配置分析（P1200-P1299）

物理向量信号端点配置分析产生会在分析各个进程（<Process>）的物理向量信号端点（<Vector>）时使用。它也在分析芯片的物理信号端点时被使用（<Chip><Vector>，见后文）可能产生的所有报错如下表所示（P1200 -- P1299）：

表 5-15 物理向量信号端点配置分析阶段可能产生的报错信息

错误编号	错误信息	原因
P1200	Name section is missing.	向量端点名称<Name>或**错误！未定义书签。**不存在。
P1201	Name section is empty.	向量端点名称<Name>或**错误！未定义书签。**为空。
P1202	Number section is missing.	向量端点名称<Name>或**错误！未定义书签。**不存在。
P1203	Number is not a valid unsigned integer.	向量端点名称<Name>或**错误！未定义书签。**不是十进制。

5.6.2.14 内核功能配置分析（P1300-P1399）

内核功能配置分析会在分析各个进程（<Process>）或虚拟机（<Virtual>）的内核功能（<Kernel>）时使用。可能产生的所有报错如下表所示（P1300 -- P1399）：

表 5-16 内核功能配置分析阶段可能产生的报错信息

错误编号	错误信息	原因
P1300	Name section is missing.	功能调用名称<Name>或**错误！未定义书签。**不存在。
P1301	Name section is empty.	功能调用名称<Name>或**错误！未定义书签。**

错误编号	错误信息	原因
		为空。
P1302	Starting kernel function number section is missing.	主功能号起始值<Begin>或**错误！未定义书签。**不存在。
P1303	Starting kernel function number is not a valid hex integer.	主功能号起始值<Begin>或**错误！未定义书签。**不是十六进制。
P1304	Ending kernel function number section is missing.	主功能号结束值<End>或**错误！未定义书签。**不存在。
P1305	Ending kernel function number is not a valid hex integer.	主功能号结束值<End>或**错误！未定义书签。**不是十六进制。
P1306	Ending kernel function number is smaller than the starting kernel function number.	结束值项比起始值项大，这是不允许的。结束值项至少应当等于起始值项。

5.6.3 芯片 XML 文件分析错误（D0000-D9999）

该模块分析芯片描述 XML 文件。这些文件是编译器自带的。下面将分章节讲解每个分步骤的具体内容。

5.6.3.1 芯片信息初步分析（D0000-D0099）

在本步骤中，系统中的额外内存信息、共享内存信息和一些工程基本信息会经分析后直接填充在内部数据结构中。可能产生的所有报错信息如下（D0000 -- D0099）：

表 5-17 芯片信息初步分析阶段可能产生的报错信息

错误编号	错误信息	原因
D0000	Chip XML parsing failed.	芯片 XML 文件存在语法错误，无法解析。
D0001	Chip XML is malformed.	芯片 XML 中不存在<Chip>标签。
D0002	Class section is missing.	名称<Name>不存在。
D0003	Class section is empty.	名称<Name>为空。
D0004	Compatible variant section is missing.	兼容列表<Compatible>不存在。
D0005	Compatible variant section is empty.	兼容列表<Compatible>为空。
D0006	Vendor section is missing.	厂商<Vendor>不存在。
D0007	Vendor section is empty.	厂商<Vendor>为空。
D0008	Platform section is missing.	平台架构<Platform>不存在。
D0009	Platform section is empty.	平台架构<Platform>为空。

错误编号	错误信息	原因
D0010	Cores section is missing.	**错误！未定义书签。**不存在。
D0011	Cores is not an unsigned integer.	**错误！未定义书签。**为空。
D0012	Regions section is missing.	通用内存保护区域数<Region>不存在。
D0013	Regions is not an unsigned integer.	通用内存保护区域数<Region>为空。
D0014	Attribute section missing.	架构属性组<Attribute>不存在。
D0015	Attribute section parsing internal error.	解析架构属性组<Attribute>及其内容时发生未知错误。请检查<Attribute>标签及其内容。
D0016	Memory section is missing.	芯片 XML 中不存在<Memory>标签。
D0017	Memory section parsing internal error.	解析<Memory>标签及其内容时遇到未知错误。请检查该标签的完整性。
D0018	Memory section is empty.	<Memory>标签内部为空。
D0019	Memory type section is missing.	存储段类别<Type>不存在。
D0020	Memory type section is empty.	存储段类别<Type>为空。
D0021	Memory type is malformed.	存储段类别<Type>必须是 Code，Data 或 Device 三者之一。当前值无法识别。
D0022	No code section exists.	芯片中没有可用于代码段的内存。
D0023	No data section exists.	芯片中没有可用于数据段的内存。
D0024	No device section exists.	芯片中没有可用于设备段的内存。
D0025	Option section is missing.	芯片 XML 中不存在<Option>标签。
D0026	Option section parsing internal error.	解析<Option>标签及其内容时遇到未知错误。请检查该标签的完整性。
D0027	Vector section missing.	芯片 XML 中不存在<Vector>标签。
D0028	Vector section parsing internal error.	解析<Vector>标签及其内容时遇到未知错误。请检查该标签的完整性。

5.6.3.2 芯片内存分析（P0500-P0599）

本步骤分析芯片本身的内存存储器。本部分所可能产生的报错（P0500 -- P0599）已经在前面完成介绍，因此在这里我们将其略去。

5.6.3.3 芯片选项分析（D0100-D0199）

本步骤分析芯片本身的选项。可能产生的所有报错信息如下（D0100 -- D0199）：

表 5-18 芯片选项分析阶段可能产生的报错信息

错误编号	错误信息	原因
D0100	Name section is missing.	**错误！未定义书签。**不存在。
D0101	Name section is empty.	**错误！未定义书签。**为空。
D0102	Type section is missing.	**错误！未定义书签。**不存在。
D0103	Type section is empty.	**错误！未定义书签。**为空。
D0104	Type is malformed.	**错误！未定义书签。**必须是 Select 或 Range 两者之一。当前值无法识别。
D0105	Macro section is missing.	**错误！未定义书签。**不存在。
D0106	Macro section is empty.	**错误！未定义书签。**为空。
D0107	Range section is missing.	**错误！未定义书签。**不存在。
D0108	Range section is empty.	**错误！未定义书签。**为空。
D0109	Range section have an empty value.	**错误！未定义书签。**之中出现了一个空值。空值是指两个逗号之间没有内容，或者第一个逗号之前没有内容，或者最后一个逗号之后没有内容。
D0110	Range typed option cannot have more or less than two ends specified.	Range 类型的选项的**错误！未定义书签。**之中必须正好有两个值标注其起止点。多或者少都是不允许的。
D0111	Starting point or ending point conversion failure.	某 Range 类型选项的**错误！未定义书签。**中有不是整数的数值。请检查其起止点确实为无符号整数。
D0112	Range typed option starting point must be smaller than the corresponding ending point.	Range 类型的选项的**错误！未定义书签。**的起点值必须严格小于止点值，否则选项将无可选择。
D0113	Select typed option's possible values must all be valid C identifiers or numbers.	Select 类型的选项的**错误！未定义书签。**的所有可能值必须全部是合法的 C 标识符，或者是一个数字。

5.6.3.4 芯片向量分析（P1200-P1299）

本步骤分析芯片本身的中断向量。本部分所可能产生的报错（P1200 -- P1299）已经在前面完成介绍，因此在这里我们将其略去。

5.6.4 内存分配错误（M0000-M0999）

该步骤确定各个自动分配的内存段的首地址，以及决定页表的具体构造方式。它又可以分成内存对齐、内存段分配、页表生成三个子部骤。

5.6.4.1 内存对齐（A0000-A9999）

该步骤主要检查并且对齐各个内存块。给 RME 和 RVM 保留的内存必须被对齐到某个由目标架构决定的粒度。各个线程的栈也要对齐到这个粒度。各个内存块的起始地址和大小也会被检查，如果有不符合该对齐粒度的情况出现，那么将会报错。这些报错随着各个架构都有所不同，具体所产生的错误需要参见该架构的相应文档。

5.6.4.2 内存段分配（M0000-M0099）

该步骤将那些没有指定起始地址的内存段分配固定的内存地址。在这一步骤中，我们先将外部内存段添加进芯片的内存布局中并检查是否有交错；如果没有，那么接下来决定各个未分配的代码和数据内存段的起始地址。

分配器使用一个高度智能化的算法决定各个内存段的位置。分配结束后，我们将检查每个进程的内存布局，确定没有交错，以及确定各个进程的主代码段是互相独立的[1]。最后，我们还要检查所有的用户指定了初始地址和长度的那些内存段都落在芯片内存布局中。

可能产生的所有报错信息如下（M0000 -- M0099）：

表 5-19 内存段分配阶段可能产生的报错信息

错误编号	错误信息	原因
M0000	Chip internal and/or extra memory section overlapped.	芯片的内存布局有重叠。比如，添加的额外内存和芯片本身提供的内存发生了交叠，这在物理上是不可实现的。
M0001	RME code section is invalid, either wrong range or wrong attribute.	RME 的代码段位置不正常，因为在芯片的内存布局中这个位置不允许存放代码，或者不可读、不可执行或不可静态映射。
M0002	RVM code section is invalid, either wrong range or wrong attribute.	RVM 的代码段位置不正常，因为在芯片的内存布局中这个位置不允许存放代码，或者不可读、不可执行或不可静态映射。
M0003	Code section is invalid, either wrong range or wrong attribute.	某进程的代码段或共享代码段位置不正常，因为在芯片的内存布局中这个位置不允许存放代码，或者无法满足该代码段声明时的那些访问权限。

[1] 主数据段则无该要求

错误编号	错误信息	原因
M0004	Code memory fitter failed.	代码段分配失败。这可能是由于使用了过多的代码存储，或者声明的代码段数过多导致代码内存碎片化。
M0005	RME data section is invalid, either wrong range or wrong attribute.	RME 的数据段位置不正常，因为在芯片的内存布局中这个位置不允许存放代码，或者不可读、不可写或不可静态映射。
M0006	RVM data section is invalid, either wrong range or wrong attribute.	RVM 的代码段位置不正常，因为在芯片的内存布局中这个位置不允许存放数据，或者不可读、不可写或不可静态映射。
M0007	Data section is invalid, either wrong range or wrong attribute.	某进程的数据段或共享数据段位置不正常，因为在芯片的内存布局中这个位置不允许存放代码，或者无法满足该数据段声明时的那些访问权限。
M0008	Data memory fitter failed.	数据段分配失败。这可能是由于使用了过多的数据存储，或者声明的数据段数过多导致数据内存碎片化。
M0009	Name is duplicate or invalid.	同一种类型的共享内存重名或者同一进程内部对同一共享内存的引用重复。
M0010	Entry not found in global database.	进程或内对共享内存的引用在共享内存的声明中找不到。
M0011	Attributes not satisfied by global entry.	进程内对共享内存的引用
M0012	Private and/or shared memory section overlapped.	在同一进程内部发生了地址空间交叠。这可能是由于两段被指定了起始地址和长度的私有内存发生交叠，也有可能是由于共享内存和私有内存发生交叠。
M0013	Process primary code section overlapped.	不同进程间的私有主代码空间发生了地址交叠。这是不允许的，因为主代码空间对于各个进程而言是独占的。
M0014	Device memory have wrong attributes.	芯片内存布局中的设备段无法满足该设备内存声明的访问权限。
M0015	Device memory segment is out of bound.	声明的设备内存无法在芯片的内存布局中找到。这可能是由于芯片的这个位置不允许存放设备内存。

5.6.4.3 页表生成（A0000-A9999）

该步骤将生成该工程中各个进程和虚拟机的页表。该部分完全是由具体的架构所决定的，因此产生的具体报错信息需要查看该架构对应的手册。

5.6.5 权能表分配错误（M1000-M1999）

该步骤自动分配各个内核对象的权能表权能号。权能号包括两个，一个是在 RVM 中创建该内核对象时使用的全局权能号，另一个是在各个进程中引用这些权能时使用的本地权能号。它又可以分为合规检查、本地权能号分配、全局权能号分配、权能宏名分配和权能配对五个步骤。

5.6.5.1 合规检查（M1000-M1099）

该步骤主要进行内核对象的不能在解析 XML 时确定的较为复杂的参数检查。可能产生的所有报错信息如下（M1000 -- M1099）：

表 5-20 合规检查阶段可能产生的报错信息

错误编号	错误信息	原因
M1000	Total number of kernel priorities must be a multiple of word size.	内核态优先级的总数必须是机器字长的整数倍。不是整数倍的值不允许。
M1001	Kernel memory allocation granularity order must be no less than 2^3=8 bytes.	内核内存分配粒度级数必须大于等于 3。
M1002	Kernel memory allocation granularity order must be no more than 2^5=32 bytes.	内核内存分配粒度级数必须小于等于 5。
M1003	There are neither virtual machines nor processes in the system.	系统中既没有虚拟机也没有原生进程。这是不允许的。
M1004	Virtual machine exists but the total number of virtual machine priorities is set to 0.	系统中存在虚拟机，但是虚拟机优先级数量却被设置为 0。这是不允许的。
M1005	Total number of virtual machine priorities must be a multiple of word size.	虚拟机优先级的总数必须是机器字长的整数倍。不是整数倍的值不允许。
M1006	Virtual machine exists but the total number of virtual event sources is set to 0.	系统中存在虚拟机，但是虚拟机事件源的数量却被设置为 0。这是不允许的。
M1007	Total number of virtual event sources must be a multiple of word size.	虚拟机事件源的总数必须是机器字长的整数倍。不是整数倍的值不允许。
M1008	Total number of virtual event sources cannot exceed 1024.	虚拟机事件源的总数量不能超过 1024。
M1009	Total number of virtual event to interrupt mappings cannot be smaller than the virtual	总计的事件源或物理中断到虚拟机虚拟向量的映射允许上限必须大于虚拟机事

错误编号	错误信息	原因
	event source number.	件源的总数量。
M1010	Priority must be bigger than service daemons' priority (4).	用户进程中的线程的优先级必须大于各个服务守护线程的优先级，也即必须大于4。
M1011	Priority must be smaller than safety daemon priority (Kern_Prios-1).	用户进程中的线程的优先级必须小于安全守护线程的优先级，也即必须小于内核优先级总数-1。
M1012	Priority must be smaller than total number of virtual machine priorities.	虚拟机的优先级必须小于虚拟机优先级的总数。
M1013	Extra capacity cannot be larger than the architectural limits.	额外权能表的表项数量不能超过架构允许的数量。在 32 位机器上这个限制为 128；在 64 位机器上这个限制为 32768。
M1014	Kernel function number range exceeded architectural limits.	内核功能的功能号超过了架构允许的范围。在 32 位机器上它的最大值为 2^{16}，在 64 位机器上它的最大值为 2^{32}。
M1015	Name is duplicate or invalid.	系统中存在重复的内核对象名称。有可能是进程的名称重复，也有可能是进程内同一类内核对象的名称重复。
M1016	Name/process name is duplicate or invalid.	同一进程内部的迁移调用入口或者信号发送端点声明的目标的名称和所在进程都相同。这是不允许的。
M1017	Name/number is duplicate or invalid.	中断向量的名称或者中断号重复。这是不允许的。

5.6.5.2 本地权能号分配（M1100-M1199）

该步骤主要分配各个进程的本地权能表中的权能号。可能产生的所有报错信息如下（M1100 -- M1199）：

表 5-21 本地权能号分配阶段可能产生的报错信息

错误编号	错误信息	原因
M1100	Capability table size exceeded architectural limits.	本地权能表的大小超过了架构允许的现值。这是不允许的。

5.6.5.3 全局权能号分配（M1200-M1299）

该步骤主要分配 RVM 在创建各个内核资源时的全局权能号。本阶段不会产生任何报错信息。

5.6.5.4 权能宏名分配（M1300-M1399）

该步骤主要分配供引用权能使用的权能宏名。本阶段不会产生任何报错信息。

5.6.5.5 权能配对（M1400-M1499）

该步骤会将信号发送端点和迁移调用入口分别与信号接收端点和迁移调用目标配对。如果编译器发现无法配对的信号发送端点和迁移调用入口，则会报错。该步骤也同时检查所有声明的物理向量名称是否能够在芯片 XML 中被找到，如果不能则报错。可能产生的所有报错信息如下（M1200 -- M1299）：

表 5-22 权能配对阶段可能产生的报错信息

错误编号	错误信息	原因
M1400	Invalid process name.	信号发送端点或迁移调用入口引用的进程不存在。
M1401	Invalid invocation name.	迁移调用入口引用的迁移调用目标不存在。
M1402	Invalid receive endpoint name.	信号发送端点引用的信号接收端点不存在。
M1403	Invalid vector number.	物理向量端点的向量号和芯片 XML 中记载的不一致。
M1404	Invalid vector name.	物理向量端点引用的向量名称在芯片 XML 中无记载。

5.6.6 内核对象分配错误（M2000-M2999）

该步骤分配各个内核对象的内核内存地址，以及各个进程内部的线程、迁移调用目标的内存布局。它又可以分为内核对象内存试分配、内核对象内存分配、内核其他内存分配、用户态库内存分配和进程内存分配五个步骤。

5.6.6.1 内核对象内存试分配和分配（M2000-M2099）

由于在开始分配内核对象时用户态库的主权能表的大小是未知的，因此需要经过一个试分配阶段。试分配阶段将分配所有的内核对象的内存地址，但不分配主权能表所占的大小。在决定了内核对象的数量后，主权能表的大小随即决定，然后在进行一轮分配得到各个内核对象的真实地址。可能产生的所有报错信息如下（M2000 -- M2099）：

表 5-23 内核对象试分配和分配阶段可能产生的报错信息

错误编号	错误信息	原因
M2000	RVM capability table size exceeded the architectural limit.	RVM 的权能表大小超过了架构允许的长度。请考虑减少 RVM 的额外权能表槽位数，或降低系统的复杂性。

5.6.6.2 内核其他内存分配（M2100-M2199）

本步骤分配内核中存在的其他内存，诸如内核栈等等。某些内存段是内核运行所必需的，另外一些内存段则是内核与用户态库交流所必需的。可能产生的所有报错信息如下（M2100 -- M2199）：

表 5-24 内核其他内存分配阶段可能产生的报错信息

错误编号	错误信息	原因
M2100	RME data section is not big enough, unable to allocate vector flags.	RME 的数据段不够大，无法分配向量标志位内存。请考虑将 RME 的数据段修改得大一些。
M2101	RME data section is not big enough, unable to allocate event flags.	RME 的数据段不够大，无法分配向量标志位内存。请考虑将 RME 的数据段修改得大一些。
M2102	RME data section is not big enough, unable to allocate kernel stacks.	RME 的数据段不够大，无法分配向量标志位内存。请考虑将 RME 的数据段修改得大一些。
M2103	RME data section is not big enough, unable to allocate kernel object memory.	RME 的数据段不够大，无法分配向量标志位内存。请考虑将 RME 的数据段修改得大一些。

5.6.6.3 用户态库内存分配（M2200-M2299）

本步骤分配用户态库内存，诸如各个守护线程所使用的栈等等。可能产生的所有报错信息如下（M2200 -- M2299）：

表 5-25 用户态库内存分配阶段可能产生的报错信息

错误编号	错误信息	原因
M2200	RVM data section is not big enough, unable to allocate init thread stack.	RVM 的数据段不够大，无法分配启动守护线程栈。请考虑将 RVM 的数据段修改得大一些。
M2201	RVM data section is not big enough, unable to allocate safety daemon thread stack.	RVM 的数据段不够大，无法分配安全守护线程栈。请考虑将 RVM 的数据段修改得大一些。
M2202	RVM data section is not big enough, unable to allocate virtual machine monitor	RVM 的数据段不够大，无法分配超调用守护线程栈。请考虑将 RVM 的数据段修改得大一些。

错误编号	错误信息	原因
	daemon thread stack.	
M2203	RVM data section is not big enough, unable to allocate vector handling daemon stack.	RVM 的数据段不够大，无法分配向量守护线程栈。请考虑将 RVM 的数据段修改得大一些。
M2204	RVM data section is not big enough, unable to allocate timer handling thread stack.	RVM 的数据段不够大，无法分配时间守护线程栈。请考虑将 RVM 的数据段修改得大一些。

5.6.6.4 进程内存分配（M2300-M2399）

本步骤分配进程本身的内存，诸如各个线程和迁移调用目标的栈等等。如果该进程实际上是一个虚拟机，那么还要分配与虚拟机监视器相关的内存。可能产生的所有报错信息如下（M2300 -- M2399）：

表 5-26 用户态库内存分配阶段可能产生的报错信息

错误编号	错误信息	原因
M2300	Data section size is not big enough, unable to allocate stack.	进程的数据段不够大，无法分配线程或迁移调用目标的栈。请考虑将进程的数据段修改得大一些。
M2301	Data section size is not big enough, unable to allocate virtual machine interrupt flags.	虚拟机的数据段不够大，无法分配虚拟向量标志内存区。请考虑将虚拟机的数据段修改得大一些。
M2302	Data section size is not big enough, unable to allocate virtual machine parameters.	虚拟机的数据段不够大，无法分配超调用参数区。请考虑将虚拟机的数据段修改得大一些。
M2303	Data section size is not big enough, unable to allocate virtual machine registers.	虚拟机的数据段不够大，无法分配虚拟化寄存器区。请考虑将虚拟机的数据段修改得大一些。

5.6.7 内核工程生成错误（G0000-G0999）

该步骤生成 RME 内核的工程。它又可以分为文件夹结构生成、配置头文件生成、启动序列源文件生成、用户可编辑源文件生成、架构相关文件生成五个步骤。

在文件夹结构生成步骤，内核所需的各个文件夹将被建立，还将填充内核源码到文件夹内。

在配置头文件生成步骤，生成选择架构所使用的配置头文件。

在启动序列源文件生成步骤，用于在启动时创建内核对象的头文件和源文件将被建立。

在用户可编辑源文件生成步骤，用户可编辑源文件将被建立。用户可编辑源文件主要负责内核态中断向量的填充。

在架构相关文件生成阶段，相关工具链或 IDE 的工程将被建立，从而将各个生成的源文件组织起来成为可编译的工程。

总的而言，本步骤及其五个子部骤都没有独立的报错信息。

5.6.8 用户态库工程生成错误（G1000-G1999）

该步骤负责生成 RVM 用户态库的工程。它又可以分为文件夹结构生成、配置头文件生成、启动序列源文件生成、架构相关文件生成四个步骤。

在文件夹结构生成步骤，用户态库工程所需的各个文件夹将被建立，还将填充用户态库源码到文件夹内。

在配置头文件生成步骤，生成选择架构所使用的配置头文件。

在启动序列源文件生成步骤，用于在启动时创建内核对象的头文件和源文件将被建立。

在用户可编辑源文件生成步骤，用户可编辑源文件将被建立。用户可编辑源文件主要负责进一步初始化以及某些可以定制的处理逻辑的编写。

在架构相关文件文件生成阶段，相关工具链或 IDE 的工程将被建立，从而将各个生成的源文件组织起来成为可编译的工程。

总的而言，本步骤可能产生的报错信息如下（G1000 -- G1999）：

表 5-27 用户态库工程生成时可能产生的报错信息

错误编号	错误信息	原因
G1000	Internal capability table computation failure.	内部权能表大小计算错误，无法生成内核对象创建代码。出现此错误说明编译器遇到内部故障，请向我们报告此故障。

5.6.9 用户进程工程生成错误（G2000-G2999）

该步骤负责生成各个用户进程的工程。它又可以分为文件夹结构生成、进程源文件生成、架构相关文件生成三个步骤。

在文件夹结构生成步骤，进程工程所需的各个文件夹将被建立，还将填充进程调用系统服务所需的库源码到文件夹内。

在进程源文件生成步骤，进程中声明的可在本地引用的各个内核对象的宏定义将会被给出，方面用户引用；另外，用户声明的线程和迁移调用目标的空函数也将被创建，等待用户填充。

在架构相关文件生成步骤，关工具链或 IDE 的工程将被建立，从而将各个生成的源文件组织起来成为可编译的工程。

总的而言，本步骤可能产生的报错信息如下（G2000 -- G2999）：

表 5-28 用户进程工程生成时可能产生的报错信息

错误编号	错误信息	原因
G2000	Virtual machine should not have threads other than Vect and User.	在虚拟机中只允许存在两个线程，一个是负责运行用户程序的 User 线程，另一个是负责接收中断的 Vect 线程。出现此错误说明编译器遇到内部故障，请向我们报告此故障。

5.6.10 工作空间生成错误（G3000-G3999）

该步骤负责生成工作空间。生成的各个子项目都将被置于工作空间之下，便于统一管理和编译。它又可以分为文件夹结构生成和架构相关文件生成两个步骤。

在文件夹结构生成步骤，工作空间的文件夹将被建立。

在架构相关文件生成步骤，我们则直接生成和架构相关的工作空间文件。

总的而言，本步骤及其两个子部骤都没有独立的报错信息。

5.6.11 文件系统操作错误（F0000-F9999）

在上面提到的工程生成步骤中，大量使用了文件系统读写操作。这些文件读写操作可能产生的报错信息如下（F1000 -- F1999）：

表 5-29 文件系统操作时可能产生的报错信息

错误编号	错误信息	原因
F0000	Folder creation failed.	在输出文件夹中创建文件夹遇到问题。检查输出文件夹是否允许读写。
F0001	Cannot open source file.	不能打开作为输入的源文件。检查输出文件夹是否允许读写。
F0002	Cannot read file.	不能打开文件进行读取。检查输出文件夹是否允许读写。
F0003	Windows/Linux stat failed.	用来判别文件大小的 stat 系统调用失败。检查文件系统相关配置是否正确。
F0004	Cannot open destination file.	无法打开作为输出的目标文件。检查输出文件夹是否允许读写。

5.6.12 架构相关文件生成错误（A0000-A9999）

本章提到的所有生成步骤中独有架构相关文件的生成。这些文件的生成是架构相关的，因此其报错需要查看架构相关手册来判明。

5.7 工程生成器图形前端

由于输入 XML 来生成系统仍然十分麻烦并且容易出错，因此本编译器也提供一个图形化用户界面。藉由本章节对图形界面模块的描述，我们也介绍编译器的功能。

5.7.1 启动图形前端

打开编译器的可执行文件并等待其加载完毕。在加载完毕后，如下的界面会弹出，用户需要选择生成工程的位置、使用的微控制器的架构以及微控制器本身的型号。

在用户选择完毕后，编译器正式启动，弹出其工作界面。工作区可以分为三个部分，分别是配置组选择区、配置项选择区和配置内容区。关于各个区域的用途和使用方式我们将在下面的章节分别介绍。

在下面的章节中，每一个标题都会对应展示 XML 文件中的相关域的名称。在域的名称前的标识符的意义如下：S 代表字符串，D 代表下拉选项框，C 代表多项选择框，H 代表十六进制数，N 代表整数。如果环绕标识符的是圆括号“()”，那么该项是可填充也可不填充的；如果环绕标识符的是方括号“[]”，那么该项必须得到填充。

5.7.2 配置类选择区

该区域负责选择配置大类。配置大类一共有七类，分别是工程配置、硬件配置、存储配置、RME 内核配置、RVM 用户态库配置、原生进程配置和虚拟机配置。其中，前五类配置每类只有一项，后两类配置的项目多少依赖于创建的进程和虚拟机的数量而定。这些配置大类都展示在左侧第一栏中，其中最下方的第八个按钮负责在被点击后生成工程。下图展示了一个典型的配置类选择区的状态。

5.7.3 配置对象选择区

对于原生进程和虚拟机配置类，每个原生进程和虚拟机对象都有一系列配置组。此时，左侧第二栏负责展示这些原生进程或虚拟机对象。当对应的对象被选中时，该对象的配置组会在配置组选择区被展示。下图展示了一个典型的配置对象选择区的状态。

5.7.4 配置组选择区

配置组选择区负责一个配置大类下的各个配置组。对于每个配置大类，其配置组的内容都不同，请参见具体章节。下图展示了一个典型的进程配置类的配置组选择区的状态。

5.7.5 配置项展示区

配置项展示区负责具体的配置项的展示和配置。同样地，针对每个配置组，其配置项的内容也不同，需要参见具体章节。下图展示了一个典型的内存块的配置项的状态。

5.7.6 工程生成按钮

工程生成按钮在界面左边栏的最下方。点击这个按钮后，会弹出一个界面，让用户选择所需要生成的工程对应的 IDE 和地址。每一个架构都有可能支持不同的 IDE；当这两者均选定后，点击生成工程按钮，即可生成最终的工程。在手机 APP 上，工程生成的功能不提供，而是生成可以被电脑版读取的 XML 文件。工程生成的界面如下。

5.8 本章参考文献

无

第 6 章　在工程中使用 RVM

6.1　总览

本章节将描述在工程中使用 RVM 及其工程生成器的一般流程。该流程首先将功能划分到保护域中，然后编写工程描述文件以反应这一划分，最后对生成的工程做一定的调整以追加行为细节。

受限于篇幅，本章将仅描述部分重要配置项的设置；对于其他配置项，请参阅前面的各章节。

6.2　功能到保护域的划分

与裸机开发不同，在使用 RVM 的工程中，软件的各个组件将被指派到不同的保护域中，各个保护域的权限都受限以约束其内部软件的行为。一旦其中一个保护域因编码故障而崩溃，其余的保护域仍将保持运行，不会受到干扰；即便有保护域被黑客攻击劫持，黑客也只能获得该保护域的权限，无法控制整个系统。

决定如何划分软件中的各组件到合适的保护域中至关重要。至少要考虑以下三个方面：（1）组件划分的粒度，（2）组件对兼容性和实时性的要求和（3）组件对设备访问的要求。为了更好地把控系统的性能，推荐先在目标微控制器上运行各项基准测试，待测试完成后视各操作开销决定组件的划分粒度。

6.2.1　组件划分的粒度

组件划分的粒度指独立组件的大小。理论上讲，任何软件都是无限可分的，直到划分为单条指令为止，而且划分粒度越细，其保护效果越好[1]。然而，划分粒度越细，跨保护域通信就越多，而跨保护域通信是有代价的[2]。因此，推荐将组件划分为合适的大小以尽量规避这一开销。合理的组件划分需要满足高内聚、低耦合这一原则，组件内部的联系应当紧密，而组件间的联系应当稀疏。换言之，一个组件应当负责一件不可或难以分割的功能，该功能的所有状态和逻辑都位于该组件之内，且该组件对其它组件提供的接口应当简明扼要、易于利用。

在划分开始时，可以将软件的每项独立功能都尽可能细地划分为组件，然后观察划分中是否有控制流和数据流的紧密耦合。有紧密耦合的组件具备如下特征之一：

1. 两个组件间互相调用的频次远高于其它组件的平均水平，比如在一个计算循环中反复调用另一组件；一旦两个组件被放在独立的保护域内，其通信开销将无法忍受。

2. 两个组件共享某个数据结构，且都有可能对该数据结构发起难以隔离、难以原子化的写操作，比如图变换操作、数据排序等；将这两个组件放在独立的保护域内没有意义，因为一个组件的崩溃必然招致数据结构的不完整，从而导致另一个组件崩溃。

3. 两个组件可以被视为一个更好描述的天然功能的拆分，合并后该组件的描述反而更加简洁；如果是这种情况，将这两个组件分配到独立的保护域中意义可能有限，因为它们不是一个逻辑功能，不需要具备崩溃后的独立恢复能力。

---

[1] 极限划分的保护域约等于数据流安全

[2] 这些代价可以在基准测试中看到

如果发现现有划分中存在这样的紧密耦合，两个组件即可合并为一个组件；当所有紧密耦合的组件都被合并后得到的就是一个合理的组件分割。在完成初始整合后若性能仍不达预期，则可以继续合并组件，直到得到安全性和性能之间的良好权衡。

6.2.2 组件对兼容性和实时性的要求

在完成组件到保护域的划分后，需要进一步决定保护域的形式。RVM 提供两种保护域，分别是原生进程和虚拟机，两者各有优劣。

在兼容性方面，原生进程必须使用 RME 原生系统调用进行编程，对其它运行时环境的兼容性则欠佳，如需使用则必须手动移植；虚拟机则可灵活使用任何客户机 RTOS，并直接兼容该 RTOS 的一切生态环境。在实时性方面，原生进程无需虚拟机监视器作为中介，跨保护域通信性能更好，中断延迟更低，可以拥有与裸机 RTOS 相仿的性能；虚拟机则相反，跨保护域通信需要虚拟机监视器介导，中断延迟时间大约是裸机的 3-4 倍。

几个常见的例子如下表所述。

表 6-1 组件划分样例

组件名称	组件划分	原因	代表性软件
键控遥控码解析	原生进程	对实时性要求高，代码简单，需要准确的定时器嘀嗒。	PT2262/PT2272 解析 38kHz 红外解析
电力拖动驱动	原生进程	对实时性要求高，代码简单，且不依赖于 RTOS。	步进/伺服电机驱动 变频器开关管控制驱动
单线总线驱动	原生进程	对实时性要求高，代码简单，且不依赖于 RTOS。	DS18B20 ATECC608A
音频总线驱动	原生进程	对实时性要求高，代码简单，需要及时刷新音频缓冲区。	各类 I2S 声卡驱动 各类 SPDIF 输出驱动
梯形图执行器	原生进程	虽然代码复杂，但往往经过严格审查，且需要周期性地执行，必须移植到原生进程。	OpenPLC
网络协议栈	虚拟机	对实时性要求不高，但内部逻辑较复杂，需要 RTOS 支持。	LwIP uIP
图形界面	虚拟机	对实时性要求不高，且可能依赖于 RTOS 支持。	uC/GUI LVGL
文件系统	虚拟机	对实时性要求不高，且可能依赖于 RTOS 支持。	ElmChan-FatFS Yaffs
脚本解释器	虚拟机	对实时性要求不高，且可能依赖于 RTOS 支持。	Micropython、 Lua

组件名称	组件划分	原因	代表性软件
串行存储器驱动	虚拟机	该驱动虽然简单，但对实时性要求不高，可以放在虚拟机中。	AT24C02 驱动 W25Q64 驱动

6.2.3 组件对设备访问的要求

部分驱动组件可能需要访问设备，因此需要将设备暴露给该组件所在的保护域。最常见也是最简单的做法是（1）仅将需要的设备设置为对该保护域可见，这可以通过将设备的内存映射加入该保护域的内存描述来实现。如果（2）多个保护域都需要使用同一个设备，则可将该设备驱动封装在一个单独的保护域中，其它保护域可以委托它来发起传输。

对于某些有较强自主性的设备，采用方法（1）将它们直接暴露给不受信任的保护域会导致重大信息安全隐患。虽然保护域内的代码没有能力突破时空保护域，但它可以配置这些自主设备，并将它们作为跳板来访问任意内存地址，或者生成高频次中断而发起拒绝服务攻击。最常见的此类设备是直接内存访问（Direct Memory Access，DMA）和定时器；恶意代码可以操纵前者来修改内核空间，或操纵后者来生成高频中断。

对于 DMA 等会自主访问内存的硬件，推荐的方法是将其交给一个单独的受信任的保护域，其它保护域需要使用 DMA 时必须委托受信保护域进行。这虽然增加了 DMA 的延迟，但保证无法利用 DMA 来破坏隔离边界。当然，如果微控制器具备输入输出内存保护单元（I/O Memory Protection Unit，IOMPU），也可以藉此限制 DMA 的访问范围，此时不需要独立的受信任保护域。

对于定时器等会自主发起高频中断的硬件，除了将其交给受信任保护域外，还可以在其中断向量中对中断数进行节流（Throttle），一旦发现中断频次过多则将其关闭，直到用户再次打开中断为止。当然，如果微控制器具备硬件节流能力，也可以藉此限制中断的频次。

6.3 保护域间通信方法的选择

RVM 在保护域之间提供三种通信方法，分别是迁移调用、信号端点和事件源。其中，迁移调用和信号端点只能在原生进程之间使用，事件源则适用于原生进程与虚拟机以及虚拟机与虚拟机之间的通信。本小节将分情况介绍推荐的通信方法。

6.3.1 耦合紧密的原生进程

对于耦合紧密、性能敏感的的原生进程间通信，推荐使用迁移调用，它可以使一个进程中的线程迁移到另一个进程中来运行，且在进入和返回时都可通过寄存器组携带少量的数据，无需通过共享内存传送。它速度最快，但可能会造成目标进程内的并发访问：老线程迁移进入目标进程后对数据结构的访问可能会被调度器选择的信息新线程打断，此时新线程也可以迁移进入目标进程，并在老线程的访问结束之前发起同步访问。

因此，如果要使用迁移调用，则必须自行处理可能存在的并发性：加锁，或使用合理的无锁数据结构。

6.3.2 耦合松散的原生进程

对于耦合松散、性能不敏感的的原生进程间通信，推荐使用信号端点和共享内存。信号端点自身具备序列化能力，即便发送端有多个线程，接收端仍可以是单个线程，这能极大地简化通信编程。

值得注意的是，信号端点仅具备接收端单线程阻塞能力，但不具备发送端阻塞能力，也不具备接收端多线程阻塞能力。所幸，这些不具备的能力可以使用原生信号端点来实现：如果需要发送端阻塞，则可以给每个发送线程准备一个接收端点，在接收端完成服务、返回结果后向相应的发送端端点返回信号，令发送端解除阻塞；如果需要接收端多线程阻塞能力，则可使用多个接收端端点，而发送端可以在发送前判断接收端线程是否忙来决定唤醒何接收端线程。当然，这也可以通过巧妙地安排服务器和客户端的优先级实现：如果服务器的优先级比客户端高[1]，则一旦服务端被唤醒，客户端就自然立刻阻塞，上述两个问题即迎刃而解，而且客户端可以用发送的信号数目来指示自身的身份。

6.3.3 原生进程和虚拟机

原生进程到虚拟机和虚拟机到原生进程的通信必须通过信号端点实现。虚拟机到原生进程的发送和原生进程之间的发送别无二致：虚拟机中存在一个发送端点，而原生进程中存在一个接收端点。

6.3.4 虚拟机和虚拟机

6.4 生成前设置

6.4.1 总体设置

6.4.1.1 设置工程信息

6.4.1.2 设置调试选项

6.4.1.3 设置芯片信息

6.4.1.4 设置内存信息

6.4.2 内核的设置

6.4.2.1 添加存储器

6.4.2.2 内核参数设置

6.4.2.3 工具链设置

6.4.2.4 文件路径设置

6.4.2.5 编写中断向量的上半部

6.4.3 虚拟机监视器的设置

[1] 且它们都位于同一个 CPU 上，这对于 RVM 而言是天然满足的

第 7 章　向 RVM 中添加新芯片

7.1.1　RME 头文件的编写

7.1.2　RVM 头文件的编写

7.1.3　芯片描述文件的编写

第 8 章　移植 RVM 到新架构

8.1　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。

8.2　用户态库的移植

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 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。

8.2.1　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 6-1　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 6-2 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;

8.2.2 和 RME 配置项有关的、工程生成器自动填充的宏

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。工程生成器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 6-3 和 RME 配置项有关的、工程生成器自动填充的宏

宏名称	对应的 RME 宏
RVM_WORD_ORDER	RME_WORD_ORDER
RVM_REGION_FIXED	RME_PGT_RAW_USER
RVM_KOM_VA_START	RME_KOM_VA_START
RVM_KOM_VA_SIZE	RME_KOM_VA_SIZE

宏名称	对应的 RME 宏
RVM_KOM_SLOT_ORDER	RME_KOM_SLOT_ORDER
RVM_PREEMPT_PRIO_NUM	RME_PREEMPT_PRIO_NUM
RVM_PHYS_VCT_NUM	RME_RVM_PHYS_VCT_NUM
RVM_PHYS_VCTF_BASE	RME_RVM_PHYS_VCTF_BASE
RVM_PHYS_VCTF_SIZE	RME_RVM_PHYS_VCTF_SIZE
RVM_VIRT_EVT_NUM	RME_RVM_VIRT_EVT_NUM
RVM_VIRT_EVTF_BASE	RME_RVM_VIRT_EVTF_BASE
RVM_VIRT_EVTF_SIZE	RME_RVM_VIRT_EVTF_SIZE
RVM_INIT_CPT_SIZE	RME_RVM_INIT_CPT_SIZE
RVM_CAP_BOOT_FRONT	RME_RVM_CAP_BOOT_FRONT
RVM_KOM_BOOT_FRONT	RME_RVM_KOM_BOOT_FRONT
RVM_CAP_DONE_FRONT	RME_RVM_CAP_DONE_FRONT
RVM_KOM_DONE_FRONT	RME_RVM_KOM_DONE_FRONT

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，工程生成器也会自动填充它们。

8.2.3 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 6-4 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。

8.2.4 和 RVM 配置项有关的、工程生成器自动填充的宏

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。它们会被工程生成器自动填充，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 6-5 和 RVM 配置有关的宏

宏名称	作用
RVM_PREEMPT_VPRIO_NUM	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_PREEMPT_VPRIO_NUM (32U)
RVM_VIRT_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： #define RVM_VIRT_MAP_NUM (32U)

此外，还有一些和架构有关的配置宏。工程生成器会自动填充它们。

8.2.5 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。

8.2.6 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 6-6 RVM 移植涉及的汇编函数

函数	意义
__RVM_Entry	RVM 的入口点。
__RVM_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。它实际上是一个宏。
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。

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。

8.2.6.1 __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 6-7 __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。

8.2.6.2 __RVM_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，该函数也不使用。该函数的使用场景大致包括如下两类：

（1）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__RVM_Stub。其中，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。

（2）对于那些如 RISC-V 等需要提前设置 gp 等寄存器[2]的架构，如果在创建新线程时直接将线程的 pc 设置成线程入口，gp 寄存器的设置便无法提前进行了。因此，我们将一个进程内的一切入口点均设置成__RVM_Stub，并采取和 ARMv7-M 类似的策略将真正的入口点放在栈上。执行到__RVM_Stub 时先设置 gp，再从栈中取出真正的入口点，然后跳转到该入口点。

无论是上面哪种情况，一旦使用到__RVM_Stub，则需要在 RVM_Stack_Init 中实现对应的栈初始化流程。

表 6-8 __RVM_Stub 的实现

函数原型	void __RVM_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。

[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] 一种简单的、默认开启的链接时优化（Link-Time Optimization，LTO）“链接器松弛（Linker Relaxation）”要求设置这个寄存器

	ptr_t Param 线程的参数。

8.2.6.3 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[1]。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。

表 6-9 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[2]，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。如果不想手工编写该汇编函数，RVM 也提供一个通用版本_RVM_MSB_Generic，可直接将此宏定义为该函数名。

8.2.6.4 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 6-10 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)

[1] LSB 算作第 0 位

[2] 如 ARM 等

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 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。

8.2.6.5 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 6-11 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。

8.2.6.6 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 6-12 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。

8.2.7 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 6-13 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。

8.2.7.1 RVM_Putchar 的实现

本操作打印一个字符到控制台。该函数较为特殊，它无需用户主动编写。

表 6-14 RVM_Putchar 的实现

函数原型	void RVM_Putchar(char Char)
意义	输出一个字符到控制台。
返回值	无。

参数	char Char 要输出到系统控制台的字符。

8.2.7.2 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 6-15 RVM_Stack_Init 的实现

函数原型	ptr_t RVM_Stack_Init(ptr_t Stack, ptr_t Size, ptr_t* Entry, ptr_t Stub)
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 栈区的起始地址。
	ptr_t Size 栈区的大小。
	ptr_t* Entry 线程的入口。考虑到那些需要将入口点改为__RVM_Stub 的场景,该参数同时也用于输出。
	ptr_t Stub __RVM_Stub 的入口位置。该参数未必会被使用。

8.2.7.3 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 6-16 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。

8.2.7.4 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 6-17 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。

8.3 工程生成器的移植

暂时待定。

8.4 本章参考文献

无

第 9 章 附录

9.1 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。

9.1.1 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，哪怕甚至是硬件故障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；更有甚者，核心的指令集都完全不相容。将 RME 和 RVM 移植到这些平台上是有可能的，但其易用性会大大降低。如果您对这样的应用有兴趣，请联系我们。

9.2 本章参考文献

无