演进・远古・原核(五阶)

Mutatus・Protero・Prokaryon(R.V)

M5P1(Prokaryon) R4T1

轻量级实时操作系统(四版一型)

技术手册

系统特性

1.高易用性

- 以轻量易用为第一准绳构建的实时操作系统
- 容易配置开发环境,尤其是其路径
- 提供最常见的线程通信接口
- 提供模板和在不同 IDE 下创建的工程

2.高实时性

- 强实时策略的线程调度，全抢占式内核
- 不同优先级的线程之间全抢占式调度
- 相同优先级的线程之间时间片轮转式调度
- 线程创建和消灭时间复杂度 O(1)
- 内存分配和释放时间复杂度 O(1)
- 大多数系统服务的时间复杂度 O(1)

3.高可移植性

- 在所有实时系统中底层汇编语言数量最少
- 严格按照 ANSI C89 要求进行编写，保证对编译器的兼容性
- 最大限度遵循 MISRA C 标准，保证代码的可读性和规范性
- 最小系统对资源的要求极低
- 可作为客户操作系统运行在其他操作系统上

4.高执行效率

- 在不牺牲可移植性的基础上进行最大限度的优化
- 对代码关键性能部分进行调优
- 同优先级和不同优先级的进程间切换时间短

5.高可靠性

- 提供参数检查(Assert)功能，进行广泛存在的接口参数检查
- 提供调试支持，可以监视、发现和报告异常
- 经形式化验证，相当于 IEC61508 SIL、橙书 A1 级别或 EAL 7+
  *暂时未完成

目录

第一章 概述

1.1 简介

在小型物联网系统中，16 到 32 位的单片机的使用越发普遍。同时，对于开发周期和可靠性的要求也在增长。因此，有必要开发轻量级的小型系统以实现简单高效的物联网应用。同时，在现代系统中，轻量级虚拟化功能的重要性逐渐增加，因此我们也需要一个轻型操作系统来作为其他操作系统的客户机使用。此外，对于此类操作系统，由于缺乏内存保护单元的介入，因此需要进行形式化验证来保证其可靠性。

RMP 实时操作系统是一种简单高效易用的全抢占式实时操作系统。它提供了典型 RTOS 内核所提供的所有特性：全抢占静态优先级时间片轮转调度器，简单的存储管理功能，简明的通信机制以及针对硬件的特殊优化能力。RMP 操作系统被设计为可以在仅具 2kB ROM、1kB RAM 的微控制器上高效地运行。

本手册从用户的角度提供了 RMP 的内核 API 的描述。在本手册中，我们先简要回顾关于小型实时操作系统的若干概念，然后分章节介绍 RMP 的特性和 API。

1.1.1 设计目的和指标

RMP 操作系统的设计目的是创造一个简单易用的开源 RTOS 内核。这个内核要具有同级别中最好的易用性，并且在效率上和主流内核相近。

1.1.2 软件版权与许可证

综合考虑到不同应用对开源系统的不同要求，RMP 内核本身所采用的许可证为 LGPL v3，但是对一些特殊情况（比如安防器材和医疗器材）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 易混术语表

在本书中，容易混淆的基本术语规定如下：

1.1.3.1 操作系统

指运行在设备上的执行最底层处理器、内存等硬件资源管理的基本软件。

1.1.3.2 线程

线程指操作系统中拥有一个独立执行栈和一个独立指令流的可被调度的实体。一个进程内部可以拥有多个线程，它们共享一个进程地址空间。

1.1.3.3 静态分配

指在系统编译时就决定好资源分配方式的分配方式。

1.1.3.4 动态分配

指在系统运行过程中，可以更改资源分配的分配方式。

1.1.3.5 软实时

指绝大多数情况下操作应该在时限之内完成，但也允许小部分操作偶尔在时限之外完成的实时性保证。

1.1.3.6 硬实时

指所有操作都必须在时限之内完成的实时性保证。

1.1.3.7 常数实时

指所有操作对用户输入和系统配置都是 O(1)的，而且执行都能在某个有实际意义的常数时限之内完成的实时性保证。这是所有实时保证中最强的一种。

1.1.3.8 对（某值）常数实时

指所有操作和响应在（某值）不变的时候都是 O(1)的，而且执行都能在某个有实际意义的常数时限之内完成的实时性保证。

1.1.4 主要参考系统

调度器部分参考了 FreeRTOS（@Amazon）。

形式化证明参考了 seL4（@2016 Data61/CSIRO）。

API 接口的实现参考了 RMProkaron（@EDI）。

其他部分的实现参考了 RT-Thread（@EDI）。

其他各章的参考文献和参考资料在该章列出。

1.2 前言

操作系统是一种运行在设备上的执行最底层处理器、内存等硬件资源管理的基本软件。对于实时操作系统而言，系统的每个操作都必须是正确的和及时的，其执行时间必须是可预测的。总的而言，有两类实时操作系统：第一类是软实时系统，第二类是硬实时系统。对于软实时系统，只要在大多数时间之内，程序的响应在时限之内即可；对于硬实时系统，系统的相应在任何时候都必须在时限之内。实际上，很少有实时应用或操作系统完全是软实时的或者完全是硬实时的；他们往往是软实时部分和硬实时部分的有机结合。一个最常见的例子是 LCD 显示屏人机界面部分是软实时的，而电机控制部分是硬实时的。

RMP 系统是一种基本实时系统。它是初步展现了实时系统的基本特性的最小系统。它一般运行在高档 8/16 位机以及低档 32 位机上面，需要一个系统定时器。它没有用户态和内核态的区别，但是可以将 MMU 和 MPU 配置为保护某段内存。它需要简单的架构相关汇编代码来进行上下文堆栈切换。若要使得其在多种架构上可运行，修改这段汇编代码是必须的。移植往往还涉及系统定时器，堆栈切换，中断管理和协处理器管理。此类系统可以使用定制的链接器脚本也可以不使用，通常而言在涉及到内存保护的时候必须使用定制的链接器脚本。

在此类系统中任务表现为线程。任务函数有可能是可重入的。每个任务会使用单独的执行栈。任务代码和内核代码可以编译在一起也可以不编译在一起。任务调用系统函数往往使用直接的普通函数调用，没有经由软中断进行系统调用的概念。

此类系统具备优先级的概念，并且一般实现了不同优先级之间的抢占和相同优先级之间的时间片轮转调度。此类系统具备初级的内存管理方案，并且这种内存管理方案一般基于 SLAB 和伙伴系统。

中断对操作系统可以是完全透明的，此时操作系统并不需要知道中断是否已经到来；如果需要在中断中进行上下文切换，那么就必须将堆栈切换汇编插入该中断函数中，此时需要用汇编代码编写中断的进入和退出。

典型的此类操作系统包括 RMProkaron、RT-Thread、FreeRTOS、uC/OS、Salvo 和 ChibiOS。对于此类实时系统，要进行完全的形式化验证通常是容易的，因为内核代码的数量非常少。

传统的软件工程最多只能系统性地对软件的功能进行测试，最多能说明软件存在缺陷的可能性很小；而形式化验证可以根据某种规范或者逻辑推演规则，证明系统符合一定的属性，也即相当于证明了在给定条件下系统不可能有缺陷。

1.3 实时操作系统及其组件的性能指标

当前市场上有几百种不同种类的 RTOS 存在，而爱好者和个人开发的内核更是数不胜数。这些系统的性能往往是良莠不齐的。我们需要一些指标来衡量这些 RTOS 的性能。下面所列的指标都只能在处理器架构相同，编译器、编译选项相同的情况下进行直接比较。如果采用了不同的架构、编译器或编译选项，得到的数据没有直接意义，只具有参考性而不具有可比性。一个推荐的方法是使用工业实际标准的 ARM 或 MIPS 系列处理器配合 GCC -O2 选项进行评估。此外，也可以使用 Chronos 模拟器配合 GCC -O2 来进行评估。在评估时还要注意，系统的负载水平可能会对这些值有影响，因此只有在系统的负载水平一致的情况下，这些值才能够被比较。

1.3.1 内核大小

内核的尺寸是衡量 RTOS 的一个重要指标。由于 RTOS 通常被部署在内存极度受限的设备中，因此内核的小体积是非常关键的。内核的尺寸主要从两个方面衡量，一是只读段大小，二是数据段大小。只读段包括了内核的代码段和只读数据段，数据段包括了内核的可读写数据段大小。在基于 Flash 的微控制器系统中，只读段会消耗 Flash，而数据段则会消耗 SRAM。[1]

由于 RTOS 是高度可配置的，其内核大小往往不是固定的，而是和所选用的配置紧密相关的。因此，衡量此项性能，应该查看衡量最小内核配置、常见内核配置和最大内核配置下的内核大小。[1]

内核大小数据的获得非常简单，只要用编译器编译该内核，然后使用专门的二进制查看器（如 Objdump）查看目标文件各段的大小即可。

1.3.2 执行时间、最坏执行时间及其抖动

执行时间指 RTOS 系统调用的用时大小。最坏执行时间指执行时间在最不利条件下能达到的最大长度。RTOS 的最坏执行时间通常会在如下情况下达到：执行最长的系统调用，并在此过程中产生了大量的缓存未命中和快表未命中。RTOS 在执行系统调用时一般都会关中断；最坏执行时间通常是系统关中断最长的时间，因此对系统的实时性的影响是非常巨大的。

最坏执行时间可以分成两类：第一类是内核系统调用的最坏执行时间，另一类是线程间同步的最坏执行时间。

要获得第一类最坏执行时间，可以在调用某个系统调用之前，计时器记下此时的时间戳 Ts，然后在系统调用结束之后，再调用计时器记下此时的时间戳 Te。然后，连续调用两次计时器，记下两个时间戳 Tts 和 Tte，得到调用计时器的额外代价为 Tte-Tts。此时，执行时间就是 Te-Ts-(Tte-Tts)。最坏执行时间，就是所有的系统调用测试之中，执行时间最大的那一个。

要获得第二类最坏执行时间，可以在通信机制的发送端调用一次计时器，记下此时的时间戳 Ts，在通信机制的接收端调用一次计时器，记下此时的时间戳 Te。对于调用计时器的代价测量是类似第一类最坏执行时间的。最终得到的 Te-Ts-(Tte-Tts)就是执行时间。最坏执行时间，就是所有的通信测试之中，执行时间最大的那一个。

执行时间的抖动也是非常重要的。在多次测量同一个系统的执行、通信时间时，我们往往会得到一个分布。这个分布的平均值是平均执行时间，其标准差（有时我们也使用极差）

被称为执行时间抖动。

对于一个 RTOS，我们通常认为执行时间、最坏执行时间和抖动都是越小越好。执行时间又可以详细分成以下几类：[1]

1.3.2.1 线程切换时间

从一个线程切换到另外一个线程所消耗的时间。我们用下图的方法进行测量。在测量时，除了使用 Te-Ts 的方法，也可以使用两次 Ts 之间的差值除以 2（下同）。线程切换包括两种情况，一种情况是同优先级线程之间互相切换，另外一种是由低优先级线程唤醒高优先级线程。[2]

在第一种情况下，我们假设图中的两个线程是相同优先级的，而且在测量开始时，我们正执行的是刚刚由线程 B 切换过来的线程 A。

进程 1：线程 A	进程 1：线程 B

永久循环
{
>　计时 Ts；
　　切换到线程 B；
}
	永久循环
	{
	　　计时 Te；
	>　切换到线程 A；
	}

在第二种情况下，我们假设图中的线程 B 优先级较高，而且在测量开始时，我们正执行的是刚刚由线程 B 切换过来的线程 A。

进程 1：线程 A	进程 1：线程 B

永久循环
{
>　计时 Ts；
　　唤醒 B；
}
	永久循环
	{
	　　计时 Te；
	>　睡眠；
	}

1.3.2.2 线程间异步通信时间

不同线程之间发送和接收异步信号所用的总时间。我们用下图的方法进行测量。我们假设线程 B 已经在接收端阻塞，线程 A 进行发送，而且线程 B 的优先级比线程 A 高。[2]

进程 1：线程 A	进程 1：线程 B

永久循环
{
>　计时 Ts；
　　向 B 线程发送；
}
	永久循环
	{
	　　计时 Te；
	>　从自己的邮箱接收；
	}

1.3.3 中断响应时间、最坏中断响应时间及其抖动

中断响应时间指从中断发生到 RTOS 调用中断对应的处理线程之间的时间。最坏中断响应时间指中断响应时间在最不利条件下能达到的最大长度。最坏中断响应时间通常会在如下情况下达到：在中断处理过程中发生了大量的缓存未命中和快表未命中。中断响应时间是 RTOS 最重要的指标，甚至可以说，RTOS 的一切设计都是围绕着该指标进行的。该指标是 RTOS 对外界刺激响应时间的最直接的标准。

要获得最坏中断响应时间，可以在中断向量的第一行汇编代码（不能等到 C 函数中再去调用，因为寄存器和堆栈维护也是中断响应时间的一部分）中调用计时器，得到一个时间

戳 Ts；在中断处理线程的第一行代码处调用计时器，得到一个时间戳 Te。对于计时器代价的测量同上。最终得到的 Te-Ts-(Tte-Tts)就是中断响应时间。最坏中断响应时间，就是所有的中断响应测试之中，响应时间最大的那一个。

中断响应时间的抖动也是非常重要的。在多次测量同一个系统的中断响应时间时，我们往往会得到一个分布。这个分布的平均值是平均中断响应时间，其标准差（有时我们也使用极差）被称为中断响应时间抖动。

对于一个 RTOS，我们通常认为中断响应时间、最坏中断响应时间和抖动都是越小越好。中断响应时间的测量通常如下所示[1][3]：

内核	线程 A

硬件中断向量
{
>　计时 Ts；
　从内核向异步端点 P 发送信号；
}
	永久循环
	{
	　计时 Te；
	>　从异步端点 P 接收信号；
	}

1.3.4 实际中断响应时间、最坏实际中断响应时间及其抖动

实际中断响应时间指软硬件系统从中断外部信号输入到发出 IO 操作响应之间的时间。最坏实际中断响应时间指实际中断响应时间在最不利条件下能达到的最大长度。影响实际最坏中断响应时间的因素中，除了那些能影响最坏中断响应时间的因素之外，还有对应的 CPU 及 IO 硬件本身的因素。

要获得实际中断响应时间，我们需要一些外部硬件来支持该种测量。比如，我们需要测量某系统的 I/O 的实际中断响应时间，我们可以将一个 FPGA 的管脚连接到某 CPU 或主板的输入管脚，然后将另一个管脚连接到某 CPU 或者主板上的输出管脚。首先，FPGA 向输入管脚发出一个信号，此时 FPGA 内部的高精度计时器开始工作；在 FPGA 接收到输出管脚上的信号的时候，FPGA 内部的高精度计时器停止工作。最终得到的 FPGA 内部计时器的时间就是系统的实际中断响应时间。最坏实际中断响应时间，就是所有测试之中，响应时间最大的那一个。

对于一个软硬件系统，我们通常认为实际中断响应时间、最坏实际中断响应时间和抖动都是越小越好。值得注意的是，实际最坏中断响应时间一般会大约等于最坏执行时间加上最坏中断响应时间加上系统 CPU/IO 的固有延迟。比如，某系统在 IO 输入来临时刚刚开始执行某系统调用，此时硬件中断向量无法立刻得到执行，必须等到该系统调用执行完毕才可以。等到该系统调用执行完毕时，实际的硬件中断向量才开始执行，切换到处理线程进行处理并产生输出。实际中断响应时间的测量通常如下所示[1]：

FPGA	被测系统

永久循环
{
>　发出信号并启动计时器；
　接收信号；
　停止计时器；
}
	永久循环
	{
	　从 I/O 上接收信号；
	　最简化的内部处理流程；
	　从 I/O 上输出信号；
	}

1.4 RMP 系统调用格式

系统调用是使用系统提供的功能的一种方法。对于 RMP 而言，系统调用和简单的函数调用是一样的。下面是 RMP 中所有的系统调用接口的列表。

系统调用名称	序号	意义
RMP_Yield	0	出让当前线程的处理器
RMP_Thd_Crt	1	创建一个线程
RMP_Thd_Del	2	删除一个线程
RMP_Thd_Set	3	设置一个线程的优先级和时间片
RMP_Thd_Suspend	4	悬起一个线程
RMP_Thd_Resume	5	解除一个线程的悬起
RMP_Thd_Delay	6	延时一段时间
RMP_Thd_Cancel	7	解除一个线程的延时
RMP_Thd_Snd	8	向某线程的邮箱发送数值
RMP_Thd_Snd_ISR	9	从中断向某线程的邮箱发送数值
RMP_Thd_Rcv	10	接收本线程邮箱值
RMP_Sem_Crt	11	创建一个计数信号量
RMP_Sem_Del	12	删除一个计数信号量
RMP_Sem_Pend	13	试图获取一个信号量
RMP_Sem_Abort	14	解除某线程对信号量的获取
RMP_Sem_Post	15	释放信号到某信号量
RMP_Sem_Post_ISR	16	从中断释放信号到某信号量

本章参考文献

[1] T. N. B. Anh and S.-L. Tan, "Real-time operating systems for small microcontrollers," IEEE micro, vol. 29, 2009.

[2] R. P. Kar, "Implementing the Rhealstone real-time benchmark," Dr. Dobb's Journal, vol. 15, pp. 46-55, 1990.

[3] T. J. Boger, Rhealstone benchmarking of FreeRTOS and the Xilinx Zynq extensible processing platform: Temple University, 2013.

第二章 系统内核

2.1 系统内核简介

RMP 的内核提供了一个可使用的实时操作系统应具有的最简单的功能，主要包括线程、线程延时、邮箱机制和信号量机制。如果用户需要更复杂的机制，那么可以从 RMP 提供的几种基本机制出发实现其他的机制。

本系统推荐用于 32 个优先级以下，64 个线程以下的系统。本系统理论上最多支持的线程数量和优先级数量都是无限的，但为了系统效率和易用性，不推荐使用本系统实现超过这个复杂度的嵌入式应用。如果需要实现更复杂的嵌入式应用，推荐使用 RMEukaryon（M7M1（等更高级的系统。

2.1.1 内核调度器

RMP 内核的调度器是一个全抢占式的固定优先级时间片轮转的调度器。该类调度器是最简单的实时系统调度器，它在同一个优先级内部使用时间片轮转法则进行线程调度，在不同的优先级之间则采取抢占式调度策略。和某些 RTOS 不同，RMP 的调度器永远不会关闭中断，从而保证了系统的实时性。为了提高内核的实时性，当处理器具备 CLZ 等指令时，RMP 还可以使用这些指令加速最高优先级查找。

RMP 没有内建对多核系统的支持。如果要支持多核处理器也是可能的，这需要编译多个 RMP 副本并且在每个处理器上运行一个副本，以多处理器多逻辑操作系统的方式存在，类似于 Barrelfish[1]。

2.1.2 内存管理与内存保护

RMP 提供了一个基于二级分割适配算法（Two-Level Segregated Fit, TLSF）[2,3]机制的内存分配器。这个分配器在所有 O(1)时间复杂度的内存分配器中具备较高空间效率。然而，在小型嵌入式系统中动态内存分配并不被推荐，因此应该尽量少用该分配器。RMP 不具备真正意义上的内存保护功能，各个线程之间的地址是没有隔离的，因此在编写应用程序时应当遵循编程规范。

2.1.3 应用升级与应用模块

由于本系统相对较为小型，因此不支持应用升级和应用模块。通常而言，使用本系统编译出的完整映像大小不应超过 128kB，因此是可以全部升级的。如果需要应用模块功能，那么需要用户自行实现。

2.1.4 系统启动流程

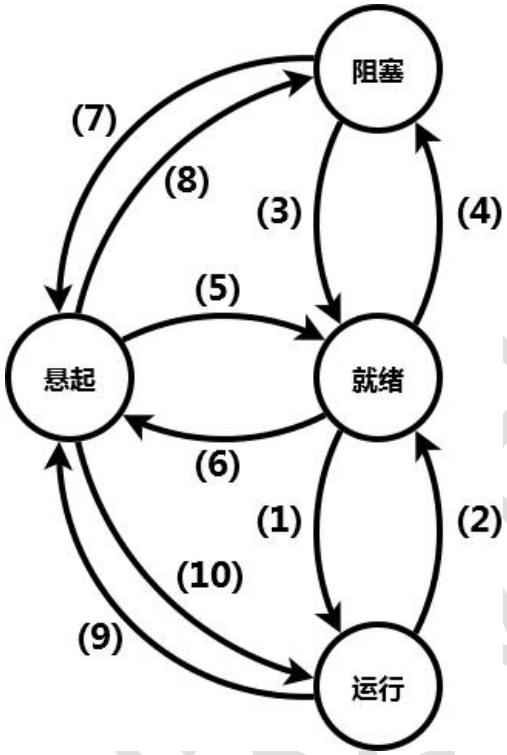
RMP 内核的第一个线程永远是 RMP_Init，它位于内核提供的代码中。Init 线程负责系统的初期初始化和低功耗，并且它永远不可能停止运行。Init 线程引出了两个接口 RMP_Init_Hook 和 RMP_Init_Idle，它们负责将用户代码导入到内核运行的过程中。关于这两个接口的说明请参见 2.4 所述。

2.2 线程相关接口

由于整个 RMP 系统都运行在一个地址空间，因此 RMP 系统仅提供线程抽象。线程可以有就绪、运行、悬起、阻塞（包括延时、等待信号量、等待邮箱接收、等待邮箱发送）四种状态，阻塞状态还具有一段时间后接收不到自动返回的功能。悬起状态可以被施加于任何

线程，此时该线程总是暂时停止执行，无论其当前处于什么状态。

完整的线程状态转移图如下所示：



图中各个数字标号的意义如下所示：

标号	代表意义
(1)	它是优先级最高的线程，因此由就绪态转入运行态。
(2)	有更高优先级的线程打断了它的执行，因此由运行态转入就绪态。
(3)	线程完成接收、发送、等待，解除阻塞，且是优先级最高的线程，因此进入运行态。
(4)	线程在接收、发送或延时功能处阻塞，因此由运行态转入等待态。
(5)	线程被解除悬起，由于其未阻塞，且不是优先级最高的线程，因此进入就绪态。
(6)	线程在就绪态被悬起，因此进入悬起态。
(7)	线程在阻塞态被悬起，因此进入悬起态。
(8)	线程被解除悬起，由于其当前仍在阻塞，因此进入阻塞态。
(9)	线程在运行态被悬起，因此进入悬起态。
(10)	线程被解除悬起，由于其未阻塞，且是优先级最高的线程，因此进入运行态。

下面是所有的线程相关接口。这些函数都只能在普通线程中调用，不能在中断服务程序中调用。它们可能有如下返回值：

返回值	数值	意义
RMP_ERR_THD	-1	由于线程控制块相关的原因导致操作失败。
RMP_ERR_PRIO	-2	由于优先级相关的原因导致操作失败。
RMP_ERR_SLICE	-3	由于时间片相关的原因导致操作失败。
RMP_ERR_STATE	-4	由于线程状态相关的原因导致操作失败。
RMP_ERR_OPER	-5	由于其他原因导致操作失败。
RMP_ERR_SEM	-6	由于信号量控制块相关的原因导致操作失败。

具体的各个函数中返回值的意义请在相应函数下查找。

2.2.1 出让处理器

本操作会使当前线程放弃 CPU，并且调度器会自动选择下一个线程进行调度。如果当前线程是唯一的优先级最高的线程，那么该线程仍然会被选中进行调度。

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

2.2.2 线程创建

本操作会创建一个新的线程并使其处于就绪态。RMP 并不提供内核对象的管理，因此线程控制块所需的内存需要用户自行分配。

函数原型	ret_t RMP_Thd_Crt(volatile struct RMP_Thd* Thread, ptr_t Entry, ptr_t Stack, ptr_t Arg, ptr_t Prio, ptr_t Slices)
返回值	ret_t 如果成功，返回 0。如果失败则可能有如下返回值：
	RMP_ERR_PRIO：输入的优先级不小于 RMP_MAX_PREEMPT_PRIO。
	RMP_ERR_SLICE：输入的时间片为 0 或不小于 RMP_MAX_SLICES。
	RMP_ERR_THD：线程控制块为 0（NULL）或者正在被使用。
参数	volatile struct RMP_Thd* Thread 指向空的、将被用于该线程的线程控制块的指针。
	ptr_t Entry 线程的入口地址。
	ptr_t Stack 线程的运行栈。对于栈向下生长的体系结构，它指向栈空间的高地址端；对于栈向上生长的体系结构，它指向栈空间的低地址端。
	ptr_t Arg 传递给线程的参数。 ptr_t Prio 线程的优先级 ptr_t Slices 线程的时间片数量。时间片数量的单位是时钟嘀嗒。

2.2.3 线程删除

本操作会删除系统中的一个线程。线程被删除后，一切资源将释放，正在向它发送的线程的发送等待将被解除并会返回发送失败。

函数原型	ret_t RMP_Thd_Del(volatile struct RMP_Thd* Thread)
返回值	ret_t 如果成功，返回 0。如果失败则可能有如下返回值：
	RMP_ERR_THD：线程控制块为 0（NULL）或者未被使用。
参数	volatile struct RMP_Thd* Thread 指向要删除的的线程的线程控制块的指针。

2.2.4 设置线程属性

本操作会设置某个线程的时间片和优先级。当只需要设置其中一个时，只要保证另一个参数与原来的相同即可。

函数原型	ret_t RMP_Thd_Set(volatile struct RMP_Thd* Thread, ptr_t Prio, ptr_t Slices)
返回值	ret_t 如果成功，返回 0。如果失败则可能有如下返回值：
	RMP_ERR_PRIO：输入的优先级不小于 RMP_MAX_PREEMPT_PRIO。
	RMP_ERR_SLICE：输入的时间片为 0 或不小于 RMP_MAX_SLICES。
	RMP_ERR_THD：线程控制块为 0（NULL）或者未被使用。
参数	volatile struct RMP_Thd* Thread 指向要更改优先级和时间片的线程的线程控制块的指针。
	ptr_t Prio 要设置的优先级。
	ptr_t Slices 要设置的时间片数量。

2.2.5 悬起线程

本操作会使某线程悬起，使调度器暂时停止对其的调度。无论线程处于哪个状态，它都可以被悬起，但是已经处于的延时、发送、接收等状态仍继续存在，也即线程处于悬起状态和延时、发送、接收等状态之一的叠加。当延时、发送、接收等状态结束后，如果该线程还处于被悬起状态，那么它将被置于纯粹的悬起状态，仍然不参与调度，直到悬起状态被手动解除。

函数原型	ret_t RMP_Thd_Suspend(volatile struct RMP_Thd* Thread)
返回值	ret_t 如果成功，返回 0。如果失败则可能有如下返回值：
	RMP_ERR_THD：线程控制块为 0（NULL）或者未被使用。
	RMP_ERR_STATE：该线程已经被悬起，无法被再次悬起。
参数	struct RMP_Thd* Thread 指向要悬起执行的线程的线程控制块的指针。

2.2.6 解除线程悬起

本操作会解除某个线程的悬起状态，使其有重新参与调度的可能。

函数原型	ret_t RMP_Thd_Resume(volatile struct RMP_Thd* Thread)
返回值	ret_t 如果成功，返回 0。如果失败则可能有如下返回值：
	RMP_ERR_THD：线程控制块为 0（NULL）或者未被使用。
	RMP_ERR_STATE：该线程未被悬起，无法被解除悬起。
参数	volatile struct RMP_Thd* Thread 指向要解除悬起的线程的线程控制块的指针。

2.2.7 线程延时

本操作会使当前线程延时一段时间。

函数原型	ret_t RMP_Thd_Delay(ptr_t Slices)
返回值	ret_t

	如果成功，返回 0。如果失败则可能有如下返回值：
	RMP_ERR_SLICE：输入的时间片为 0 或不小于 RMP_MAX_SLICES。
	RMP_ERR_OPER：该延时没有进行完毕就被解除。
参数	ptr_t Slices 要延时的时间，单位是时间片。

2.2.8 解除线程延时

本操作会解除某个线程的延时。

函数原型	ret_t RMP_Thd_Cancel(volatile struct RMP_Thd* Thread)
返回值	ret_t 如果成功，返回 0。如果失败则可能有如下返回值：
	RMP_ERR_THD：线程控制块为 0（NULL）或者未被使用。
	RMP_ERR_STATE：该线程未处于延时状态，无法被解除延时。
参数	volatile struct RMP_Thd* Thread 指向要解除延时的线程的线程控制块的指针。

2.3 线程间通信相关接口

RMP 提供了相对简单但高效的线程间通信机制，包括了线程邮箱和信号量。这些机制可以被单独使用，也可以被组合成更复杂的通信接口使用。可以阻塞的通信接口都有三个选项：探知可能阻塞后立即返回、阻塞超时后返回、一直阻塞。这大大提高了接口使用的灵活性。需要注意的是，所有的通信接口在有多个线程阻塞时，服务顺序永远是按照时间先后顺序的，不提供优先级插队功能。如果一个高优先级线程在低优先级线程阻塞后才阻塞，那么第一个被服务的是低优先级线程，然后才是高优先级线程。

2.3.1 向线程邮箱发送

本操作会向线程邮箱发送一个处理器字长的信息。

函数原型	ret_t RMP_Thd_Snd(volatile struct RMP_Thd* Thread, ptr_t Data, ptr_t Slices)
返回值	ret_t 如果成功，返回 0。如果失败则可能有如下返回值：
	RMP_ERR_THD：线程控制块为 0（NULL）或者未被使用。
	RMP_ERR_OPER：在不阻塞条件下探测到可能造成阻塞，或者试图发送到自己的邮箱，或者发送因超时、目标线程被删除而失败。
参数	volatile struct RMP_Thd* Thread 指向要发送到的线程的线程控制块的指针。
	ptr_t Data 要发送的数据。
	ptr_t Slices 要等待的时间片数量。如果为 0，那么意味着如果探测到阻塞则立即返回；如果为 0 到 RMP_MAX_SLICES 之间的数值（不包括 RMP_MAX_SLICES），那么将会最多阻塞等待该数量的时间片后返回；如果为超过或等于 RMP_MAX_SLICES 的值，那么意味着将永远阻塞直到被接收或者目标线程被销毁。

2.3.2 在中断中向线程邮箱发送

本操作会从中断向量中向线程邮箱发送一个处理器字长的信息。和上面的普通版本调用不同，该版本总是立即返回而不会阻塞。

函数原型	ret_t RMP_Thd_Snd_ISR(volatile struct RMP_Thd* Thread, ptr_t Data)
返回值	ret_t 如果成功，返回 0。如果失败则可能有如下返回值：
	RMP_ERR_THD：线程控制块为 0（NULL）或者未被使用。
	RMP_ERR_OPER：目标线程的邮箱已满，无法发送，或者调度器被锁。
参数	volatile struct RMP_Thd* Thread 指向要发送到的线程的线程控制块的指针。
	ptr_t Data 要发送的数据。

2.3.3 从线程邮箱接收

本操作会从当前线程的邮箱中接收一个值。

函数原型	ret_t RMP_Thd_Rcv(ptr_t* Data, ptr_t Slices)
返回值	ret_t 如果成功，返回 0。如果失败则可能有如下返回值：
	RMP_ERR_OPER：在不阻塞条件下探测到可能造成阻塞，或者接收因超时而失败。
参数	ptr_t* Data 该参数用于输出，输出接收到的数据。 ptr_t Slices 要等待的时间片数量。如果为 0，那么意味着如果探测到阻塞则立即返回；如果为 0 到 RMP_MAX_SLICES 之间的数值（不包括 RMP_MAX_SLICES），那么将会最多阻塞等待该数量的时间片后返回；如果为超过或等于 RMP_MAX_SLICES 的值，那么意味着将永远阻塞直到接收完成。

2.3.4 创建信号量

本操作会创建一个新的信号量。

函数原型	ret_t RMP_Sem_Crt(volatile struct RMP_Sem* Semaphore, ptr_t Number)
返回值	ret_t 如果成功，返回 0。如果失败则可能有如下返回值：
	RMP_ERR_SEM：信号量控制块为 0（NULL）或者已被使用。
	RMP_ERR_OPER：初始信号量值过大，超出或等于 RMP_SEM_MAX_NUM。
参数	volatile struct RMP_Sem* Semaphore 指向空的、将被用于该信号量的信号量控制块的指针。
	ptr_t Number 该信号量的初始值，应该小于 RMP_SEM_MAX_NUM；

2.3.5 删除信号量

本操作会删除一个信号量。 如果有线程在其上阻塞，那么该线程会直接返回信号量获取失败。

函数原型	ret_t RMP_Sem_Del(volatile struct RMP_Sem* Semaphore)

返回值	ret_t 如果成功，返回 0。如果失败则可能有如下返回值：
	RMP_ERR_SEM：信号量控制块为 0（NULL）或者未被使用。
参数	volatile struct RMP_Sem* Semaphore 指向要删除的信号量的信号量控制块的指针。

2.3.6 等待信号量

本操作会使当前线程尝试获取信号量。获取的数量总是为 1。

函数原型	ret_t RMP_Sem_Pend(volatile struct RMP_Sem* Semaphore, ptr_t Slices)
返回值	ret_t 如果成功，返回当前信号量的剩余值。如果失败则可能有如下返回值：
	RMP_ERR_SEM：信号量控制块为 0（NULL）或者未被使用。
	RMP_ERR_OPER：在不阻塞条件下探测到可能造成阻塞，或者等待因超时、目标信号量被删除、被中途解除而失败。
参数	volatile struct RMP_Sem* Semaphore 指向要等待的信号量的信号量控制块的指针。
	ptr_t Slices 要等待的时间片数量。如果为 0，那么意味着如果探测到阻塞则立即返回；如果为 0 到 RMP_MAX_SLICES 之间的数值（不包括 RMP_MAX_SLICES），那么将会最多阻塞等待该数量的时间片后返回；如果为超过或等于 RMP_MAX_SLICES 的值，那么意味着将永远阻塞直到获取到信号量或者目标信号量被销毁。

2.3.7 解除信号量等待

本操作解除某线程的信号量等待。如果等待成功被解除，目标线程的等待信号量函数会返回 RMP_ERR_OPER。

函数原型	ret_t RMP_Sem_Abort(volatile struct RMP_Thd* Thread)
返回值	ret_t 如果成功，返回 0。如果失败则可能有如下返回值：
	RMP_ERR_THD：线程控制块为 0（NULL）或者未被使用。
	RMP_ERR_STATE：该线程未等待信号量状态，无法被解除等待。
参数	volatile struct RMP_Thd* Thread 指向要取消信号量等待的线程的线程控制块的指针。

2.3.8 发布信号量

本操作会发布一定数量的信号到某信号量。

函数原型	ret_t RMP_Sem_Post(volatile struct RMP_Sem* Semaphore, ptr_t Number)
返回值	ret_t 如果成功，返回 0。如果失败则可能有如下返回值：
	RMP_ERR_SEM：信号量控制块为 0（NULL）或者未被使用。
	RMP_ERR_OPER：目标信号量如果接受此数量的信号会溢出，也即超过 RMP_SEM_MAX_NUM。
参数	volatile struct RMP_Sem* Semaphore 指向要发布到的信号量的信号量控制块的指针。

	ptr_t Number 要发布的信号数量。

2.3.9 从中断发布信号量

本操作会从中断中发布一定数量的信号到某信号量。

函数原型	ret_t RMP_Sem_Post_ISR(volatile struct RMP_Sem* Semaphore, ptr_t Number)
返回值	ret_t 如果成功，返回 0。如果失败则可能有如下返回值：
	RMP_ERR_SEM：信号量控制块为 0（NULL）或者未被使用。
	RMP_ERR_OPER：目标信号量如果接受此数量的信号会溢出，也即超过 RMP_SEM_MAX_NUM，或者调度器被锁。
参数	volatile struct RMP_Sem* Semaphore 指向要发布到的信号量的信号量控制块的指针。
	ptr_t Number 要发布的信号数量。

2.4 内存管理接口

RMP 提供了一个基于 TLSF 的内存分配器。它可以有效地管理动态内存。TLSF 分配器是一种二级适配算法，它把内存块先按照 2 的方次放入不同的初级指数区间（First-Level Interval，FLI），如 127-64 Byte，255-128 Byte 等，然后在每个区间之内再将内存块放入各个线性区间，比如在 127-64 Byte 的二级线性区间（Second-Level Interval，SLI）就有(127, 112]，(111, 96]，……，(79, 64]一共 8 个。在分配时，先查找某大小对应的 SLI 的下一级 SLI 有没有可供分配的内存块，如果有则分配；如果没有则向上查找到最近的 SLI，从那里分配内存。当内存被释放时，内存块会被即刻合并并且放入对应的 SLI，准备下次分配。

RMP 的 TLSF 实现中，FLI 的数目会根据内存池的大小决定，SLI 的数目则固定为 8，同时规定当 FLI=0 时对应 127-64Byte 的内存块，并且单个内存池的大小不能低于 4096 个机器字（对于 16 位机不能少于 8kB，对于 32 位机则不能少于 16kB），不能多于 128MB。由于 RMP 普遍被用于 128kB 以下内存的环境，因此要管理的内存大小很少能超过这个限制。此外，内存池和内存池的大小均要求对其到机器字，并且一次分配的最小内存数量为 64Byte。

内存池的初始化和使用均非常灵活（可以每个线程使用自己的私有内存池也可以任意共享），因此 RMP 的内存池操作默认是不锁调度器的。如果需要锁调度器，那么可以使用 RMP_Lock_Sched 和 RMP_Unlock_Sched 自行实现。

最后，要注意 RMP 的内存分配器并不实现内存保护功能，无法在内存结构被破坏时恢复。因此，使用动态分配的内存时一定要注意不要在被分配的区域外进行读写。如果进行读写，可能会破坏分配器的数据结构而引起异常。

RMP 的 TLSF 分配器仅仅包括三个调用，如下列出。

2.4.1 初始化内存池

本操作按照传入的内存池大小和地址初始化该内存池。内存池的大小和地址必须对齐到机器字。

函数原型	ret_t RMP_Mem_Init(volatile void* Pool, ptr_t Size)
返回值	ret_t 如果成功，返回 0。如果失败则可能有如下返回值：

	RMP_ERR_MEM：内存池为空（NULL）或大小、地址未对齐，或者传入的内存池大小小于 4096 个机器字，或者传入的内存池大小大于 128MB。
参数	volatile void* Pool 指向要初始化为内存池的空白内存的指针。
	ptr_t Size 该块空白内存的大小。

2.4.2 从内存池分配内存

本操作会试图从某内存池分配内存。分配的最小数量为 64Byte，如果试图分配的数量小于它而大于 0，那么实际上会分配 64Byte；如果传入的大小为 0 则会直接返回分配失败。

函数原型	void* RMP_Malloc(volatile void* Pool, ptr_t Size)
返回值	void* 如果成功，指向返回内存的非 0（NULL）指针。如果失败则返回 0（NULL）。
参数	volatile void* Pool 指向要分配内存的内存池的指针。
	ptr_t Size 要分配的内存大小，单位是 Byte。

2.4.3 向内存池归还内存

本操作会向某个内存池归还内存。传入的指针必须是由 RMP_Malloc 返回的地址完全相同的地址，并且必须归还到分配该内存时使用的内存池。

函数原型	void RMP_Free(volatile void* Pool, void* Mem_Ptr)
返回值	无。
参数	volatile void* Pool 指向要归还内存的内存池的指针。
	void* Mem_Ptr 指向要归还的内存块的指针。

2.5 其他系统接口

RMP 提供的其他接口包括了中断开关、调度器锁定解锁和一些辅助函数。这些函数可以在用户编写应用程序的时候提供一些便利。

2.5.1 中断系统和调度器系统接口

在 RMP 中，中断系统的接口仅包括一对开关中断的函数。这一对函数是提供给用户使用的，因为 RMP 自身在运行过程中不关闭中断。除非硬性需要，否则不建议关闭中断，因为这会大大影响系统的实时性。

调度器系统的接口则包括一对带有嵌套计数功能的使能和除能调度器的函数。这一对函数也提供给用户使用。由于锁调度器也会降低系统的实时性，因此也不建议经常使用。总的方针是，如果不锁调度器能解决问题，那就不锁调度器；如果锁调度器和关中断都能解决，那么锁调度器；只有必须关中断时才关中断。

2.5.1.1 除能中断

本操作关闭处理器对所有中断源的响应，包括系统定时器中断和调度中断。该函数不具

备嵌套计数功能；如果需要嵌套计数功能，那么需要用户自行实现。

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

2.5.1.2 使能中断

本操作开启处理器对所有中断源的响应。该函数不具备嵌套计数功能；如果需要嵌套计数功能，那么需要用户自行实现。

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

2.5.1.3 锁定调度器

本操作锁定调度器，使调度器无法选择新的线程进行调度。该函数具备嵌套计数功能。

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

2.5.1.4 解锁调度器

本操作解锁调度器，使之又可以选择新的线程进行调度。该函数具备嵌套计数功能。

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

2.5.2 辅助库函数接口

为了方便用户应用程序的编写，RMP 提供了一系列库函数供用户使用。这些库函数包括了内存清零、调试信息打印和链表操作等。这些库函数的列表如下：

2.5.2.1 清零内存

本操作将一段内存清零。

函数原型	void RMP_Clear(volatile void* Addr, ptr_t Size)
返回值	无。
参数	volatile void* Addr 要清零的内存段的起始地址。
	ptr_t Size 要清零的内存段的大小，单位为字节。

2.5.2.2 打印一个字符

本操作打印一个字符到控制台（一般是串口）。

函数原型	void RMP_Putchar(char Char)
返回值	无。
参数	char Char 要打印的字符本身。

2.5.2.3 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。

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

2.5.2.4 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。

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

2.5.2.5 打印字符串

本操作打印一个最长不超过 255 字符的字符串到调试控制台。

函数原型	cnt_t RMP_Print_String(s8* String)
返回值	cnt_t 返回打印的字符数量。
参数	s8* String 要打印的字符串。

2.5.2.6 得到一个字的最高位

本操作得到一个字的最高位的位号（比如对于 32 位处理器会返回 0-31）。如果该数字为 0，那么需要返回-1（对于 32 位处理器为 0xFFFFFFFF）。

函数原型	ptr_t RMP_MSB_Get(ptr_t Val)
返回值	ptr_t 返回该字最高位的位号。
参数	ptr_t Val 要求出最高位位号的无符号整数。

2.5.2.7 得到一个字的最低位

本操作得到一个字的最低位的位号（比如对于 32 位处理器会返回 0-31）。如果该数字为 0，那么返回一个负值。

函数原型	ptr_t RMP_LSB_Get(ptr_t Val)
返回值	ptr_t 返回该字最低位的位号。
参数	ptr_t Val 要求出最低位位号的无符号整数。

2.5.2.8 翻转一个字的高低位

本操作会将一个字的比特位翻转，例如在 32 位处理器上会令第 0 位和第 31 位交换，第

1 位和第 30 位交换，第 2 位和第 29 位交换，依此类推。

函数原型	ptr_t RMP_RBIT_Get(ptr_t Val)
返回值	ptr_t 高低位翻转后的该字。
参数	ptr_t Val 要翻转高低位的无符号整数。

2.5.2.9 创建双向循环链表

本操作初始化双向循环链表的链表头。

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

2.5.2.10 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。

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

2.5.2.11 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。

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

2.5.3 钩子函数接口

为了方便在系统运行的一些关键点插入用户所需的功能，RMP 系统还提供了钩子函数。要使用这些钩子函数，需要定义宏 RMP_USE_HOOKS 为 RMP_TRUE。系统中所提供的钩子函数的列表如下：

2.5.3.1 系统启动钩子

该钩子函数会在系统完成处理器初始化后立即被调用。

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

2.5.3.2 系统上下文保存钩子

该钩子函数会在系统完成基本上下文保存后被调用。如果有额外的上下文（FPU 和其他外设寄存器组）需要保存，可以在该函数中将这些上下文压栈。具体的实现方法请参见第四章。

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

2.5.3.3 系统上下文恢复钩子

该钩子函数会在系统完成基本上下文恢复前被调用。如果有额外的上下文（FPU 和其他外设寄存器组）需要恢复，可以在该函数中将这些上下文弹栈。具体的实现方法请参见第四章。

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

2.5.3.4 系统嘀嗒钩子

该钩子函数会在系统嘀嗒到来时被调用。该函数有一个参数，是指向当前经过的 Tick 数量的指针。因此该函数可以读取也可以修改当前经过的 Tick 的数量，这可以用来实现无嘀嗒系统。具体的无嘀嗒系统实现方法请参见第四章。

函数原型	void RMP_Tick_Hook(ptr_t* Ticks)
返回值	无。
参数	ptr_t* Ticks 指向当前时钟嘀嗒和上一个时钟嘀嗒之间经过的嘀嗒数量的指针。

2.5.3.5 空闲线程首次运行钩子

该钩子函数会在空闲线程首次运行时被调用一次。这个钩子函数总是被使能的，不论宏 RMP_USE_HOOKS 的定义情况如何。在这个函数中可以添加更多的线程创建操作以启动其他线程，或者做一些系统初始化工作。

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

2.5.3.6 空闲线程反复运行钩子

该钩子函数会在空闲线程被运行时反复被调用。这个钩子函数总是被使能的，不论宏 RMP_USE_HOOKS 的定义情况如何。典型的用法是在这个函数中进入低功耗模式以省电，或者做一些性能分析、堆栈检测打印输出。

函数原型	void RMP_Init_Idle(void)
返回值	无。

参数	无。

本章参考文献

[1] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, et al., "The multikernel: a new OS architecture for scalable multicore systems," in Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, 2009, pp. 29-44.

[2] M. Masmano, I. Ripoll, A. Crespo, and J. Real, "TLSF: A new dynamic memory allocator for real-time systems," in Real-Time Systems, 2004. ECRTS 2004. Proceedings. 16th Euromicro Conference on, 2004, pp. 79-88.

[3] X. Sun, J. Wang, and X. Chen, "An improvement of TLSF algorithm," in Real-Time Conference, 2007 15th IEEE-NPSS, 2007, pp. 1-5.

第三章 形式化验证

3.1 形式化验证简介

嵌入式实时操作系统的本质是复杂性、多样性和可靠性。RMP 操作系统的硬件无关部分包括大约 2000 行代码，各个部分之间具有较复杂的作用和联系方式。作为一个操作系统又要应付不同应用的各种需求，并且需要保证功能的正确性。同时，我们还需要提高开发效率，控制成本和开发周期，以及保证软件的顺利交付。

传统的软件设计方法基于自然语言的思考、设计和描述，往往片面和模糊，极易引起误解。它也无法进行严格的检查，只能通过人的心智进行分析。基于 UML 等半形式化的方法采用一些相对清晰的图形化描述，一些工具也能自动生成代码框架并检查分析。以上两种方法在测试系统时，均是设计一系列用例对其进行测试，最多有结构化测试的参与。但是，它们都无法保证系统中没有错误，均不适用于性命攸关的系统的分析和开发，因此我们需要更为严格的开发设计流程。完全的形式化方法则基于严格定义的数学概念和语言，可以开发自动化工具进行检查和分析。它把数学的严格性带入软件开发的各个阶段，通过严格的数学证明保证系统中没有漏洞。

软件安全性的 EAL 标准分为 7 个等级，分别如下（EAL7+也列入在内）：

EAL 级别	描述
EAL1	EAL1 是**经过功能性测试**的系统。它提供基本的安全保障目标描述。这些描述中包含了系统的功能规范和潜在错误分析。在 EAL1 中，每一个安全保障目标都要列出并且评估测试。 相比于未评估系统，EAL1 仅仅是一个小规模的安全提升。
EAL2	EAL2 是**使用基本软件工程方法论进行结构化开发**的系统。它提供**全面**的安全保障目标描述。这些描述中包含了系统的基本功能规范和**基本安全架构**。在 EAL2 中，每一个安全保障目标都要列出并且进行**独立的评估测试**；此外还需要有**一定攻击能力**的安全测试人员对这些**安全保障目标**进行攻击，以确定没有问题。 EAL2 相比于 EAL1 有更严格的测试规范和系统弱点分析。
EAL3	EAL3 是**使用强化的软件工程方法论进行系统性测试和检查**的系统。它提供全面的安全保障目标描述。这些描述中包含了系统的基本功能规范和**安全架构的详细设计**。在 EAL3 中，每一个安全保障目标都要列出并且进行独立的评估测试；此外还需要有一定攻击能力的安全测试人员**针对安全架构的详细设计**进行攻击，以确定没有问题。 EAL3 相比于 EAL2 有更严格的系统架构规范和相应的测试。
EAL4	EAL4 是**使用强化的软件工程方法论进行系统性设计、测试和评估**的系统。它提供全面的安全保障目标描述。这些描述中包含了系统的**详细功能规范**和安全架构的详细设计，**还要包含一部分关键模块的划分和具体实现**。在 EAL4 中，每一个安全保障目标都要列出并且进行独立评估测试；此外还需要有**高于平均水平**攻击能力的安全测试人员针对安全架构的详细设计进行攻击，以确定没有问题。 EAL4 相比于 EAL3 有更严格的设计描述、实现审查和更强大的的测试。
EAL5	EAL5 是**使用半形式化方法进行设计和测试**的系统。它提供全面的安全保障目标描述。这些描述中包含了系统的详细功能规范和安全架构的详细设计，还要包含**半形式化方法描述的全部关键模块的划分**和一部分模块的具体实现。在 EAL5 中，每一个安全保障目标都要列出并且进行独立评估测试；此外还需要有**相当程度**攻击能力的安全测试人员针对安全架构的详细设计进行攻击，以确定没有

	问题。 EAL5 相比于 EAL4 有半形式化的设计描述、更清晰的架构和更强大的测试。
EAL6	EAL6 是**使用半形式化方法进行设计、测试和验证**的系统。它提供全面的安全保障目标描述。这些描述中包含了系统的详细功能规范和安全架构的详细设计，还要包含全部关键模块的划分和**全部模块的具体实现**。EAL6 要求对于安全保障目标的策略有**完整的形式化描述**，对其功能则要有**半形式化描述**。具体的功能设计必须是**模块化的、分层的、和简明的**。在 EAL6 中，每一个安全保障目标都要列出并且进行独立评估测试；此外还需要有**较强**攻击能力的安全测试人员针对安全架构的详细设计进行攻击，以确定没有问题。 EAL6 相比于 EAL5 有更全面的设计分析和弱点测试、以及结构化的实现描述。
EAL7	EAL7 是**形式化方法进行设计、测试和验证**的系统。它提供全面的安全保障目标描述。这些描述中包含了系统的详细功能规范和安全架构的详细设计，还要包含全部关键模块的划分和**全部模块的结构化的具体实现**。EAL7 要求对于安全保障目标的策略有完整的形式化描述，对其功能则要有半形式化描述。具体的功能设计必须是模块化的、分层的、和简明的。在 EAL6 中，每一个安全保障目标都要列出并且**针对实现进行完整的独立评估测试**；此外还需要有**极强**攻击能力的安全测试人员针对安全架构的详细设计进行攻击，以确定没有问题。 EAL7 相比于 EAL6 有更进一步的形式化设计和更全面的测试。
EAL7+	EAL7+是**严格形式化方法进行设计、测试和验证**的系统。它提供季全面的安全保障目标描述。这些描述中包含了系统的详细功能规范和安全架构的详细设计，还要包含全部关键模块的划分和全部模块的结构化的具体实现。EAL7+要求对于安全保障目标的功能和策略有**完整的形式化描述**。具体的功能设计必须是模块化的、分层的、和简明的，**并且要证明具体的实现完全符合形式化描述本身**。在 EAL7+中，每一个安全保障目标都要列出并且针对实现进行完整的独立评估测试，**测试用例必须由形式化工具自动生成**；此外还需要有极强攻击能力的安全测试人员针对安全架构的详细设计进行攻击，以确定没有问题。 EAL7+相比于 EAL7 有更进一步的形式化实现和形式化测试。 **RMP 系统是采用 EAL7+标准设计和测试的。**

绝大多数操作系统如 RT-Thread、FreeRTOS、Windows、Linux 的认证级别都在 EAL4+。少数几款其他系统的认证在 EAL5，INTEGRITY-178B 的认证级别在 EAL6+。由于 RMP 的结构相对简单，而且其安全目标中只包括功能正确性目标而不包括信息安全目标（CC/CAPP/LSPP 等设计规范不适用），因此较为容易取得更高的认证水平。

3.2 系统的形式化规范

在 RMP 中，由于架构相关部分非常短小（往往在 50 行代码以内），容易保证其正确性，因此不对该部分进行形式化验证。另一个原因是架构相关部分往往随芯片而发生变化，不方便进行验证。因此，主要的验证工作集中在 kernel.c 文件的 1500 行代码上。注释不计算在内，约有 1000 行之多。

3.3 形式化证明

将完成的后续工作

3.4 其他文档

将完成的后续工作

本章参考文献

[1] Common Criteria. "Common Criteria for Information Technology Security Evaluation", Part 3: Security assurance components, 2012.

第四章 移植 RMP 到新架构

4.1 移植概述

操作系统的移植是指将一个操作系统加以修改从而使其能运行在一个新的体系架构上的工作。有时，我们也把使得能用一款新编译器编译该操作系统的工作叫做移植工作。相比较于 uC/OS 和 RT-Thread 等系统的移植，RMP 的移植是非常简单的。RMP 的所有代码都用相对符合 MISRA C 规范的 ANSI/ISO C89 代码写成，并包含有最小量的汇编，因此其移植工作仅仅需要几步。

在移植之前，我们要先做一些准备工作，以确定移植可以进行；然后，分别针对各个部分，编写相应的移植代码即可。最后，还可以用一些测试用例来测试系统是否正确移植成功。由于 RMP 的非架构相关部分代码经过了形式化验证，因此不要对非架构相关部分进行任何修改，否则会造成系统认证被破坏。

4.2 移植前的检查工作

4.2.1 处理器

RMP 要求所选择的处理器具备至少两个中断向量和一个定时器。除此之外，RMP 对处理器没有其他任何要求。RMP 不能在少于 8kB 闪存存储器的平台上运行，也不能在低于 16 位的处理器上运行。如果要在这些平台上运行操作系统，采用基于状态机的 RMS 可能是一个更好的选择。如果所选择的处理器是多核的或具备 MMU，那么运行基于微内核的 RME 可能是一个更好的选择。RMP 不支持硬件堆栈机制，堆栈必须是由软件实现的（也即堆栈指针可以由用户修改，堆栈实现在内存中），而不能在处理器内部通过硬件实现（后者是 PIC 单片机等少数架构的典型实现方式）。

4.2.2 编译器

RMP 要求编译器是 C89 标准的，并能够根据一定的函数调用约定生成代码。由于 RMP 的代码非常标准，也不使用 C 运行时库中的库函数，因此只要编译器符合 ANSI C89 标准即可。通常的 gcc、clang/llvm、msvc、armcc、icc、ewxxx、tasking 等编译器都是满足这个需求的。RMP 没有使用位段、enum 和结构体对齐等各编译器实现差别较大的编译器扩展，也没有使用 C 语言中的未定义操作，因此保证了最大限度的兼容性。

在使用编译器时，要注意关闭编译器的（激进的）死代码消除功能和链接时优化功能，最好也要关闭编译器的循环不变量外提优化。不要使用任何激进的编译优化选项，在一般的编译器上，推荐的优化选项是（如 gcc）-O2 或相当的优化水平。

4.2.3 汇编器

RMP 要求汇编器能够引入 C 中的符号，并根据函数调用约定进行调用；此外，也要求汇编器产生的代码能够导出并根据函数调用约定被 C 语言调用。这通常是非常好满足的要求。如果编译器可以内联汇编，那么不需要汇编器也是可以的。

4.2.4 调试器

RMP 对调试器没有特别的要求。如果有调试器可用的话，当然是最好的，但是没有调试器也是可以进行移植的。在有调试器的情况下可以直接用调试器查看内核变量；在没有调试器的情况下，要先实现内核最底层的 RMP_Putchar 函数，实现单个字符的打印输出，然后就可以用该打印输出来输出日志了。关于该函数的实现请看下节所述。

4.3 RMP 架构相关部分介绍

RMP 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 Cortex-M 架构的文件夹名称为 Platform/CortexM。其对应的头文件在 Include/Platform/CortexM，其他架构以此类推。

每个架构都包含一个或多个源文件和一个或多个头文件。内核包含架构相关头文件时，总是会包含 Include/RMP_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 RMP 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 Cortex-M 架构进行编译，那么该头文件就应该包含对应 Cortex-M 的底层函数实现的全部头文件。

在移植到其他架构时，可以用 Cortex-M 架构的底层作为一套模板并在它的基础上展开新架构的移植工作。

4.3.1 类型定义

对于每个架构/编译器，首先需要移植的部分就是 RMP 的类型定义。对于类型定义，只需要确定处理器的字长在编译器中的表达方法，使用 typedef 定义即可。需要注意的是，对于某些架构和编译器，long（长整型）类型对应的是两个机器字的长度，而非一个机器字；此时应当使用 int 类型来表达一个机器字的长度。对于另一些架构和编译器，int 是半个机器字的长度，long 是一个机器字的长度，此时应当注意用 long 来定义一个机器字。

在必要的时候，可以使用 sizeof()运算符编写几个小程序，来确定该编译器的机器字究竟是何种标准。

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。

类型	意义
s8	一个有符号八位整形。 例如：typedef char s8;
s16	一个有符号十六位整形。 例如：typedef short s16;
s32	一个有符号三十二位整形。 例如：typedef int s32;
u8	一个无符号八位整形。 例如：typedef unsigned char u8;
u16	一个无符号十六位整形。 例如：typedef unsigned short u16;
u32	一个无符号三十二位整形。 例如：typedef unsigned int u32;

对于 RMP 而言，必须被定义的的类型定义一共有如下三个：

类型	作用
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;
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;

4.3.2 宏定义

其次，需要移植的是 RMP 的宏定义。RMP 的宏定义一共有如下几个：

宏名称	作用
EXTERN	编译器的 extern 关键字。某些编译器可能具有不标准的 extern 关键字，此时用这个宏定义来处理它。 例子：#define EXTERN extern
RMP_WORD_ORDER	处理器字长（按 Bit 计算）对应的 2 的方次。比如，32 位处理器对应 5，64 位处理器对应 6，依此类推。 例子：#define RMP_WORD_ORDER 5
RMP_INIT_STACK	初始线程堆栈起始地址。如果堆栈向下生长，这就是堆栈的顶部；如果堆栈向上生长，这就是堆栈的底部。 RMP 声明了另外两个宏 RMP_INIT_STACK_HEAD(X)和 RMP_INIT_STACK_TAIL(X)，可以利用这两个宏来实现。前者的意义是从内核中定义的初始线程堆栈头部开始往后部偏移一段距离，后者则是从后部向前部偏移一段距离，距离的单位是机器字长。 例子： #define RMP_INIT_STACK RMP_INIT_STACK_TAIL(16) 从初始堆栈数组的后部向前部偏移 16 个机器字长。
RMP_INIT_STACK_SIZE	初始线程堆栈大小，单位为字节。
RMP_MAX_PREEMPT_PRIO	内核支持的抢占优先级的最大数量。这个数量必须是处理器字长（按 Bit 计算）的整数倍。通常而言，把这个值定义为处理器字长就可以了。 例子：#define RMP_MAX_PREEMPT_PRIO 32
RMP_MAX_SLICES	内核允许的线程时间片或延时时间片的最大数量。 例子：#define RMP_MAX_SLICES 100000
RMP_SEM_MAX_NUM	内核允许的信号量计数的最大数量。 例子：#define RMP_SEM_MAX_NUM 100
RMP_USE_HOOKS	是否使用钩子函数。如果使用钩子函数，那么用户需要提供 2.4.3 节中所述的四个钩子函数的实现。 例子：#define RMP_USE_HOOKS RMP_TRUE
RMP_MASK_INT() RMP_UNMASK_INT()	是否在锁调度器时屏蔽能调用中断发送函数的中断。如果不使用该功能，这两个宏可以定义为空，此时调用中断发送函数的一系列中断仍然可能在锁调度器时发生，但它们的中断发送函数均会因调度器被锁而返回失败。如果使用该功能，RMP_MASK_INT()可以定义为掩蔽所有系统中断，而 RMP_UNMASK_INT()可以定义为解除系统中断掩蔽，这样就能保证那些调用中断发送函数的中断在锁调度器期间不发生。当然，处理器不具备该功能时也可将两宏简单定义为开关全局中断，但这样做会影响系统的实时性。 例子：#define RMP_MASK_INT　MASK(SYSPRIO) #define RMP_MASK_INT　MASK(0x00)

	更详细的例子请参看 Cortex-M3 处理器上的 RMP 移植。MASK(SYSPRIO)的意义是掩蔽所有的优先级为 SYSPRIO 及以下的中断，MASK(0x00)则为解除所有掩蔽。具体的 SYSPRIO 值应该被定义为能调用中断发送函数的中断的优先级的最高值。

4.3.3 底层汇编函数

RMP 仅要求用汇编或内联汇编实现 4 个短小的底层汇编函数。这些函数的名称和意义如下：

函数名	意义
RMP_Disable_Int	禁止处理器中断。
RMP_Enable_Int	使能处理器中断。
_RMP_Yield	触发线程切换。
_RMP_Start	启动初始线程。

这些函数的具体实现方法和实现要求将在后面章节加以讲解。

4.3.4 系统中断向量

RMP 最低仅仅要求用汇编或内联汇编实现 2 个中断向量。这些中断向量的名称和意义如下：

中断向量名	意义
系统定时器中断向量	处理系统定时器中断，管理时间片使用。
线程切换中断向量	处理线程切换时使用。

这些中断向量的具体实现方法和实现次序将在后面章节加以讲解。

4.3.5 其他底层函数

这些底层函数涉及 RMP 的启动、调试等其他方面。这些函数可以用汇编实现，也可以不用汇编实现，也可以部分使用 C 语言，部分使用内联汇编实现。这些函数的列表如下：

函数	意义
RMP_Putchar	打印一个字符到内核调试控制台。
RMP_MSB_Get	得到一个字的最高位（MSB）位置。
_RMP_Low_Level_Init	底层硬件初始化。
_RMP_Stack_Init	初始化某线程的线程栈。

4.4 底层汇编函数的移植

汇编底层函数的原型和移植要求详述如下。

4.4.1 RMP_Disable_Int 的实现

函数原型	void RMP_Disable_Int(void)
意义	关闭处理器中断。
返回值	无。
参数	无。

该函数需要关闭处理器的中断，然后返回。实现上没有特别需要注意的地方，通常而言只需要写一个 CPU 寄存器或者外设地址，关闭中断，然后返回即可。

4.4.2 RMP_Enable_Int 的实现

函数原型	void RMP_Enable_Int(void)
意义	开启处理器中断。
返回值	无。
参数	无。

该函数需要开启处理器的中断，然后返回。实现上没有特别需要注意的地方，通常而言只需要写一个 CPU 寄存器或者外设地址，开启中断，然后返回即可。

4.4.3 _RMP_Yield 的实现

函数原型	void _RMP_Yield(void)
意义	软件触发可悬起的线程切换中断向量。
返回值	无。
参数	无。

该函数需要软件触发可悬起的线程切换中断向量。通常而言这是写入某个内存地址或者执行某条特殊指令。

4.4.4 _RMP_Start 的实现

函数原型	void _RMP_Start(ptr_t Entry, ptr_t Stack)
意义	开始执行初始线程。
返回值	无。
参数	ptr_t Entry 初始线程的入口地址，实际上就是 RMP_Init。
	ptr_t Stack 初始线程的栈地址。

该函数实现从内核栈到线程栈的切换，仅在系统启动阶段的最后被调用。在此之后，系统进入正常运行状态。该函数只要将 Stack 的值赋给堆栈指针，然后直接跳转到 Entry 即可。该函数将永远不会返回。

4.5 系统中断向量的移植

RMP 系统需要移植两个中断向量，分别是系统定时器中断向量和线程切换中断向量。系统定时器中断向量没有必须使用汇编编写的要求，而线程切换中断向量则必须使用汇编编写。

4.5.1 定时器中断向量

在定时器中断处理向量中，仅仅需要调用如下函数：

函数原型	void _RMP_Tick_Handler(ptr_t Ticks)
意义	执行定时器中断处理。
返回值	无。
参数	ptr_t Ticks 在本次时钟中断与上次时钟中断之间经过的嘀嗒数。

这个函数是系统实现好的，无需用户自行实现。

4.5.2 线程切换中断向量

线程切换中断向量必须使用汇编编写，并且按顺序完成以下步骤：

1.先切换到内核栈，再将 CPU 的基本寄存器全部压入线程栈；

2.再调用 RMP_Save_Ctx 保存多余的上下文；

3.然后将现在的线程堆栈指针存入变量 RMP_Cur_SP；

4.然后调用_RMP_Get_High_Rdy 进行线程切换；

5.然后将变量 RMP_Cur_SP 赋给现在的线程堆栈指针；

6.再调用 RMP_Load_Ctx 恢复多余的上下文；

7.最后将 CPU 的基本寄存器全部从线程栈恢复，并切换到线程栈，退出中断。

其中 RMP_Save_Ctx 和 RMP_Load_Ctx 两个函数在第二章已经介绍过，在这里不再赘述。这里仅简要介绍线程切换函数。

函数原型	void _RMP_Get_High_Rdy(void)
意义	执行线程切换处理。该切换会更新变量 RMP_Cur_SP 和 RMP_Cur_Thd，供上下文切换汇编段和其他内核函数使用。
返回值	无。
参数	无。

这个函数也是系统实现好的，无需用户自行实现。

4.6 其他底层函数的移植

在剩下的底层函数中，RMP_Putchar 和 RMP_MSB_Get 在第二章已经讨论过，我们在这里仅讨论_RMP_Low_Level_Init 和_RMP_Stack_Init。

4.6.1 底层硬件初始化

函数原型	void _RMP_Low_Level_Init(void)
意义	初始化包括 PLL、CPU、中断系统、系统嘀嗒计时器在内的所有系统基本硬件。
返回值	无。
参数	无。

该函数初始化所有的底层硬件。在初始化中断系统时，应当把系统嘀嗒定时器的优先级设为最低，将线程切换中断的优先级设置得次低，并且两者不能和任何其他中断有嵌套关系。调用了 RMP_Thd_Snd_ISR 和 RMP_Sem_Post_ISR 的中断向量也不允许和其他中断向量出现嵌套关系。对于其他的中断向量，则没有此种约束，可以任意嵌套。

4.6.2 初始化某线程的线程栈

函数原型	void _RMP_Stack_Init(ptr_t Entry, ptr_t Stack, ptr_t Arg)
意义	填充线程的线程栈，以便在新线程第一次运行时模拟出返回至此处的假象。
返回值	无。
参数	ptr_t Entry 该线程的入口。
	ptr_t Stack 该线程的初始堆栈地址。
	ptr_t Arg 给该线程传入的参数。

该函数在系统中会被线程创建函数调用。由于线程切换中断的后半段会从该栈中弹出寄存器，因此这里应当按照同样的顺序放置各个寄存器，尤其是线程入口和参数。具体的序列依各个处理器而有不同，和线程切换中断中具体压栈弹栈的实现顺序也有关系。

4.6.3 无节拍内核的实现

无节拍内核通常要求系统具备一个高精度定时器，并且由该高精度定时器产生系统的调度器时间中断。此时只需要在系统上下文恢复钩子中检查一遍当前选中的线程的剩余时间，再检查一遍最近超时的定时器的剩余时间，然后选择最小的那个剩余时间，将其设置成硬件定时器的下一次超时时间即可。内核为此提供了函数_RMP_Get_Near_Ticks 用于得到这个数值。这个函数不是必须被调用的，如果不实现无节拍内核可以不必理会。如果这段时间的长度超过了硬件定时器能支持的范围，那么可以将这个时间长度分成几段来执行，比如将其保存在一个临时变量中，然后进行多轮延时。在每轮延时的定时器钩子中将其每次减去本轮延时时间，直到剩余时间可以在一轮之内用尽为止。

函数原型	ptr_t _RMP_Get_Near_Ticks(void)
意义	得到最近超时的嘀嗒数，可根据此设置系统定时器的下一次超时时间。
返回值	ptr_t 最近一次超时的嘀嗒数量。
参数	无。

这个函数是系统实现好的，无需用户自行实现。

4.6.4 浮点处理器上下文的保存和恢复

要保存和恢复浮点处理器的寄存器，应当使用 RMP_Save_Ctx 和 RMP_Load_Ctx 两个钩子。在保存钩子内，先要探测当前线程是否使用了浮点寄存器，如果是那么保存浮点寄存器到堆栈或者其他地方。在恢复钩子内也要先探测浮点寄存器是否被使用，如果是的话那么需要从刚刚保存的地方恢复它。在 RMP 上，推荐的浮点处理器使用方法是仅仅让一个线程使用浮点处理器，这样就可以不必保存和恢复浮点处理器的上下文，从而提高系统的实时性和效率。

4.6.5 低功耗设计注意事项

在低功耗设计中，推荐使用无节拍内核，此外还要在 RMP_Init_Idle 钩子内插入能让处理器进入休眠状态的指令。在 ARM 架构上，WFI 和 WFE 指令可以达到这个效果，而在 MSP430 处理器上则必须修改 STATUS 寄存器的某些位让处理器进入休眠。

本章参考文献

无