M5P01 轻量型

深度嵌入式实时操作系统

技术参考手册

版本 R6T1

演进·远古·原核（五阶）

Mutatus·Protero·Prokaron(R.V)

M5P01(Prokaron) R6T1

轻量级实时操作系统（六版一型）

技术参考手册

系统特性

高易用性

·以轻量易用为第一准绳构建的实时操作系统

·容易配置开发环境及工程路径

·提供最常见的线程通信接口

高实时性

·强实时策略的全抢占式内核

·相同优先级的线程之间时间片轮转式调度

·所有关键操作时间复杂度 O(1)

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 和 MISRA C 要求进行编写，保证对编译器的兼容性

·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

高执行效率

·对代码关键性能部分进行调优

·极短的线程切换时间和中断响应时间

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

·经形式化验证，相当于 IEC61508 SIL4

*暂时未完成，目前相当于 SIL2 预认证级别

目录

表目录

图目录

版本历史

版本	日期（年-月-日）	说明
R1T1	2018-02-15	初始发布
R2T1	2018-03-12	增加了移植说明和附录
R4T3	2018-07-31	修正手册的错误
R4T3	2018-09-01	更新手册到新版本
R4T4	2018-10-10	解除内存管理单元的 128MB 限制，使用更优化的圆角矩形绘制算法
R4T5	2018-11-25	解除最大优先级个数的限制，允许更激进的内存用量优化策略
R5T1	2020-03-01	在基本接口之外增加额外的线程通信接口
R5T2	2023-05-07	为信号量增加广播和同步解锁模式
R6T1	2024-06-26	调整钩子宏的位置和声明，调整 GUI 的配置宏规则

第 1 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（Real-Time Operating System，RTOS）。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实时性和可靠性，并且在平均执行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 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 易混术语表

在本手册中，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] 比如安防器材和医疗器材等

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情形。

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情形。

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

调度器与 API 部分参考了 FreeRTOS（@RT Engineering LTD/Amazon）。

形式化证明参考了 seL4（@2016 Data61/CSIRO）。

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。此类系统是初步展现了实时系统的基本特性的最小系统。它们一般运行在高档 8/16 位机以及低档 32 位机上面，需要一个系统定时器。此类系统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。此类系统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[1]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。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 T_s，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 T_e。然后，连续调用两次计时器，记下两个时间戳 T_{ts} 和 T_{te}，得到调用计时器的额外代价为 T_{te}-T_{ts}。此时，执行时间就是 T_e-T_s-(T_{te}-T_{ts})。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 T_s，在通信机制的接收端调用一次计时器，记下此时的时间戳 T_e。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 T_e-T_s-(T_{te}-T_{ts})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，简称执行时间；其标准差[2]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

线程切换时间指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 T_e-T_s 的方法，也可以使用两次 T_s 之间的差值除以 2。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。

表 1-1 线程切换的第一种情况

线程 A	线程 B
永久循环	永久循环

[1] 如 Objdump

[2] 有时我们也使用极差

线程 A	线程 B
{	{
>> 计时 T_s;	计时 T_e;
切换到线程 B;	>> 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。

表 1-2 线程切换的第二种情况

线程 A	线程 B
永久循环	永久循环
{	{
>> 计时 T_s;	计时 T_e;
唤醒 B;	>> 睡眠;
}	}

1.3.2.2 异步通信时间

异步通信时间指不同线程之间发送和接收异步信号所用的时间。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]。

表 1-3 线程间异步通信

线程 A	线程 B
永久循环	永久循环
{	{
>> 计时 T_s;	计时 T_e;
向 B 线程发送;	>> 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（Trans Look-aside Buffer，TLB）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[1]中调用计时器，得到一个时间戳 T_s；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 T_e。对于计时器代价的测量同上。最终得到的 T_e-T_s-(T_{te}-T_{ts})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[2]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]：

表 1-4 中断响应时间

内核	线程 A
硬件中断向量	永久循环
{	{
>> 计时 T_s；	计时 T_e；
从内核向异步端点 P 发送信号；	>> 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]：

[1] 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

[2] 有时我们也使用极差

表 1-5 实际中断响应时间

FPGA（或者示波器）	被测系统
永久循环	永久循环
{	{
>> 发出信号并启动计时器；	从 I/O 上接收信号；
接收信号；	最简化的内部处理流程；
停止计时器；	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唯一方法。对于 RMP 这样的库结构操作系统而言，系统调用和通常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。

表 1-6 RMP 系统调用一览

系统调用名称	序号	意义
RMP_Thd_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

1.5 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 2 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。本系统还额外提供了一个轻量级图形界面库，它在不使用[1]的时候完全不消耗内存。

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 RME（M7M01）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 的调度器在某些架构上永远不会关闭中断，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、FFS 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[2]内建对多核系统的直接支持。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，以分区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 线程引出了两个钩子 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 2.6 所述。

---

[1] 未检测到相关宏被定义

[2] 将来也永远不会

2.2 线程调度接口

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[1]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：

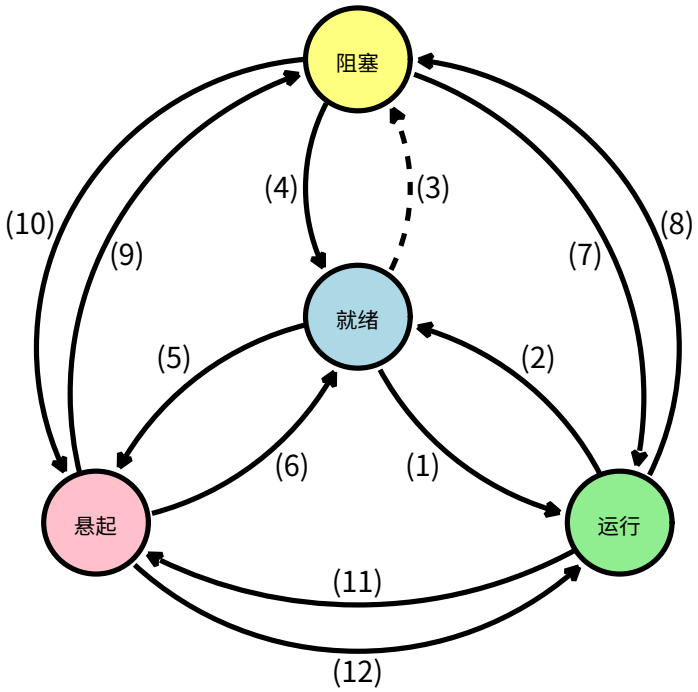

图 2-1 线程状态转移图

图中各个数字标号的意义如下所示：

表 2-1 线程状态转移图中各标号的含义

标号	代表意义
(1)	该线程处于就绪态且优先级最高，因此由就绪态转入运行态。
(2)	该线程处于运行态且优先级非最高，因此由运行态转入就绪态。
(3)	该可能性不存在（图中以虚线表示），线程不可能直接由就绪态进入阻塞态。
(4)	该线程被解除阻塞且未悬起，且优先级非最高，因此由阻塞态进入就绪态。
(5)	该线程在就绪态被悬起，因此由就绪态进入悬起态。
(6)	该线程被解除悬起且未阻塞，且优先级非最高，因此由悬起态进入就绪态。
(7)	该线程被解除阻塞且未悬起，且优先级最高，因此由阻塞态进入运行态。
(8)	该线程在邮箱、信号量或延时功能处阻塞，因此由运行态进入阻塞态。

[1] 包括延时、等待信号量、等待邮箱接收、等待邮箱发送四种基本阻塞方式

标号	代表意义
(9)	该线程在被解除悬起且仍在阻塞，因此由悬起态进入阻塞态。
(10)	该线程在阻塞态被悬起，因此由阻塞态进入悬起态。
(11)	该线程在运行态被悬起，因此由运行态进入悬起态。
(12)	该线程在被解除悬起且未阻塞，且优先级最高，因此由悬起态进入运行态。

所有的线程调度接口如下表所示：

表 2-2 线程调度接口一览

接口	意义	锁调度器时调用	中断中调用
RMP_Thd_Crt	创建线程	可	不可
RMP_Thd_Del	删除线程	可	不可
RMP_Thd_Set	设置线程属性	可	不可
RMP_Thd_Yield	出让处理器	不可[1]	不可
RMP_Thd_Suspend	悬起线程	不可	不可
RMP_Thd_Resume	解除线程悬起	可	不可
RMP_Thd_Delay	线程延时	不可	不可
RMP_Thd_Cancel	解除线程延时	可	不可
RMP_Thd_Loop	线程循环延时	可	不可

它们可能有如下返回值：

表 2-3 线程调度接口的可能返回值

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_STACK	-4	由于运行栈相关的原因导致操作失败。
RMP_ERR_STATE	-5	由于线程状态相关的原因导致操作失败。
RMP_ERR_OPER	-6	由于其他原因导致操作失败。

[1] 不可能阻塞当前线程的操作均可在锁调度器时调用，但可能会阻塞或出让当前线程的操作则不可在锁调度器时调用；这个标准同样适用于后面各章节中列出的的接口

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 创建线程

本操作会创建一个新的线程并使其处于就绪态。新创建的线程总是会被插入到同优先级时间片轮转队列的末尾：如果没有其它干扰，则先创建的线程总是先被调度。

表 2-4 创建线程

函数原型	rmp_ret_t RMP_Thd_Crt(volatile struct RMP_Thd* Thread, void* Entry, void* Param, void* Stack, rmp_ptr_t Size, rmp_ptr_t Prio, rmp_ptr_t Slice)
返回值	rmp_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_PREEMPT_PRIO_NUM。
	RMP_ERR_SLICE 输入的时间片为 0 或不小于 RMP_SLICE_MAX。
	RMP_ERR_STACK 输入的栈区域为 0（NULL）或者栈大小放不下上下文。
	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。
	void* Entry 线程的入口地址。
	void* Param 传递给线程的参数。
	void* Stack 线程的运行栈的起始地址。如果以数组的形式分配它，则此处填写数组的首地址。某些架构对栈的内存地址有特殊要求[1]，此时该地址必须满足这些要求。
	rmp_ptr_t Size 线程的运行栈的大小，单位为字节。如果以数组的形式分配它，则此处填写数组的大小。
	rmp_ptr_t Prio 线程的优先级。在本系统中，优先级数字越大，优先级越高。
	rmp_ptr_t Slice

[1] 如必须位于地址空间的低 64KiB 等；如果涉及堆栈段寄存器，请参见 6.5.3.2

	线程的时间片数量。时间片数量的单位是时钟嘀嗒。

2.2.2 删除线程

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。

值得注意的是，RMP 的线程并不允许通过返回来直接退出。如果任何一个线程想要终止自己，就必须显式调用这个函数删除自己。这个设计是有意而为，是为了显式地提醒用户线程正被删除，从而方便用户最大限度地避免“创建 - 运行一次 - 删除”的用法[1]。

表 2-5 删除线程

函数原型	rmp_ret_t RMP_Thd_Del(volatile struct RMP_Thd* Thread)
返回值	rmp_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。

2.2.3 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是 RMP_PREEMPT_PRIO_NUM 或 RMP_SLICE_MAX 即可。

表 2-6 设置线程属性

函数原型	rmp_ret_t RMP_Thd_Set(volatile struct RMP_Thd* Thread, rmp_ptr_t Prio, rmp_ptr_t Slice)
返回值	rmp_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 输入的优先级大于 RMP_PREEMPT_PRIO_NUM。
	RMP_ERR_SLICE 输入的时间片为 0 或者大于 RMP_SLICE_MAX。
	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。
	rmp_ptr_t Prio

[1] 这种用法应该被“阻塞 - 运行一次 - 再次阻塞”或线程池技术代替

	要设置的优先级。输入 RMP_PREEMPT_PRIO_NUM 代表不修改优先级。 在本系统中，优先级数字越大，优先级越高。
	rmp_ptr_t Slice 要设置的时间片数量。输入 RMP_SLICE_MAX 代表不修改时间片。

2.2.4 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。如果当前线程的优先级是最高的，且那个优先级上只有它，那么它仍然会被调度。

表 2-7 出让处理器

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

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。

表 2-8 悬起线程

函数原型	rmp_ret_t RMP_Thd_Suspend(volatile struct RMP_Thd* Thread)	
返回值	rmp_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。 该检查即使在 RMP_CHECK_ENABLE 关闭时也会执行。
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。

表 2-9 解除线程悬起

函数原型	rmp_ret_t RMP_Thd_Resume(volatile struct RMP_Thd* Thread)	
返回值	rmp_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。 该检查即使在 RMP_CHECK_ENABLE 关闭时也会执行。
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。

表 2-10 线程延时

函数原型	rmp_ret_t RMP_Thd_Delay(rmp_ptr_t Slice)	
返回值	rmp_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_SLICE_MAX。
	RMP_ERR_OPER	该延时没有进行完毕就被解除。 该检查即使在 RMP_CHECK_ENABLE 关闭时也会执行。
参数	rmp_ptr_t Slice 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。

表 2-11 解除线程延时

函数原型	rmp_ret_t RMP_Thd_Cancel(volatile struct RMP_Thd* Thread)	
返回值	rmp_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。 该检查即使在 RMP_CHECK_ENABLE 关闭时也会执行。
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。	

2.2.9 线程循环延时

本操作会使当前线程陷入死循环一段时间。与其它线程接口不同，它可以在锁调度器或者关中断的条件下使用，且延时粒度精细，但会白白浪费处理器性能。

表 2-12 线程循环延时

函数原型	void RMP_Thd_Loop(rmp_ptr_t Loop)
返回值	无。
参数	rmp_ptr_t Loop 要延时的时间，单位是循环数。循环数和处理器周期之间的换算可能随不同的处理器、工具链和优化等级而有不同。

2.3 线程邮箱和信号量接口

RMP 提供了简单而高效的线程间通信机制，包括线程邮箱（Mailbox）和信号量（Semaphore），其中信号量是带操作计数、排队等待、阻塞解锁和批量唤醒的，因而既可作为线程间的条件变量和计数器又可作为线程内的锁。上述两个基本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。

所有的线程邮箱和信号量接口如下表所示：

表 2-13 线程邮箱和信号量接口一览

接口	意义	锁调度器时调用	中断中调用
RMP_Thd_Snd	向线程邮箱发送	不可	不可
RMP_Thd_Snd_ISR	在中断中向线程邮箱发送	不可	可
RMP_Thd_Rcv	从线程邮箱接收	不可	不可
RMP_Sem_Crt	创建信号量	可	不可
RMP_Sem_Del	删除信号量	可	不可
RMP_Sem_Post	发布信号量	可	不可
RMP_Sem_Post_ISR	从中断发布信号量	不可	可
RMP_Sem_Bcst	广播信号量	可	不可
RMP_Sem_Bcst_ISR	从中断广播信号量	不可	可
RMP_Sem_Pend	等待信号量	不可	不可

接口	意义	锁调度器时调用	中断中调用
RMP_Sem_Pend_Unlock	等待信号量并解除调度器锁	可，仅单层锁	不可
RMP_Sem_Abort	解除信号量等待	可	不可
RMP_Sem_Cnt	查询信号量数目	可	不可

它们可能有如下返回值：

表 2-14 线程邮箱和信号量接口的可能返回值

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_SLICE	-3	由于时间片相关的原因导致操作失败。
RMP_ERR_STATE	-5	由于线程状态相关的原因导致操作失败。
RMP_ERR_OPER	-6	由于其他原因导致操作失败。
RMP_ERR_SEM	-7	由于信号量控制块相关的原因导致操作失败。

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。

表 2-15 向线程邮箱发送

函数原型	rmp_ret_t RMP_Thd_Snd(volatile struct RMP_Thd* Thread, rmp_ptr_t Data, rmp_ptr_t Slice)	
返回值	rmp_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。 最后一条检查即使在 RMP_CHECK_ENABLE 关闭时也会执行。
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。	
	rmp_ptr_t Data 要发送的数据。	
	rmp_ptr_t Slice 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_SLICE_MAX 之间的值，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_SLICE_MAX 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。

表 2-16 在中断中向线程邮箱发送

函数原型	rmp_ret_t RMP_Thd_Snd_ISR(volatile struct RMP_Thd* Thread, rmp_ptr_t Data)	
返回值	rmp_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。 该检查即使在 RMP_CHECK_ENABLE 关闭时也会执行。
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。	
	rmp_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处理器字长的值。

表 2-17 从线程邮箱接收

函数原型	rmp_ret_t RMP_Thd_Rcv(rmp_ptr_t* Data, rmp_ptr_t Slice)	
返回值	rmp_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。 该检查即使在 RMP_CHECK_ENABLE 关闭时也会执行。
参数	rmp_ptr_t* Data 该参数用于输出，输出接收到的数据。如果传入 RMP_NULL，则接收到的数据会被丢弃。	
	rmp_ptr_t Slice 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_SLICE_MAX 之间的值，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_SLICE_MAX 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。

表 2-18 创建信号量

函数原型	rmp_ret_t RMP_Sem_Crt(volatile struct RMP_Sem* Semaphore, rmp_ptr_t Number)	
返回值	rmp_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，超过或等于 RMP_SEM_CNT_MAX。
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。	
	rmp_ptr_t Number 该信号量的初始值，应该小于 RMP_SEM_CNT_MAX。	

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，那么这些线程的等待信号量函数会直接返回 RMP_ERR_OPER。

表 2-19 删除信号量

函数原型	rmp_ret_t RMP_Sem_Del(volatile struct RMP_Sem* Semaphore)	
返回值	rmp_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 发布信号量

本操作会发布一定数量的信号到某信号量。

表 2-20 发布信号量

函数原型	rmp_ret_t RMP_Sem_Post(volatile struct RMP_Sem* Semaphore, rmp_ptr_t Number)
返回值	rmp_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	欲发布的数量为 0，或者目标信号量如果接受此数量的信号会溢出（超过或等于 RMP_SEM_CNT_MAX）。
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。	
	rmp_ptr_t Number 要发布的信号数量。	

2.3.7 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。

表 2-21 从中断发布信号量

函数原型	rmp_ret_t RMP_Sem_Post_ISR(volatile struct RMP_Sem* Semaphore, rmp_ptr_t Number)	
返回值	rmp_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	欲发布的数量为 0，或者目标信号量如果接受此数量的信号会溢出（超过或等于 RMP_SEM_CNT_MAX）。
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。	
	rmp_ptr_t Number 要发布的信号数量。	

2.3.8 广播信号量

本操作会发布足够数量的信号到某信号量，使其上面的一切等待线程解除阻塞。如果该信号量上无线程阻塞，则不会发送任何信号。

表 2-22 广播信号量

函数原型	rmp_ret_t RMP_Sem_Bcst(volatile struct RMP_Sem* Semaphore)	
返回值	rmp_ret_t 如果成功，返回成功解除阻塞的线程的数量。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	

	指向要广播到的信号量的信号量控制块的指针。

2.3.9 从中断广播信号量

本操作会发布足够数量的信号到某信号量，使其上面的一切等待线程解除阻塞。如果该信号量上无线程阻塞，则不会发送任何信号。使用本操作可能导致一次解除大量线程的阻塞，从而造成中断执行超期；使用时必须加以注意。

表 2-23 从中断广播信号量

函数原型	rmp_ret_t RMP_Sem_Bcst_ISR(volatile struct RMP_Sem* Semaphore)	
返回值	rmp_ret_t 如果成功，返回成功解除阻塞的线程的数量。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 指向要广播到的信号量的信号量控制块的指针。	

2.3.10 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。

表 2-24 等待信号量

函数原型	rmp_ret_t RMP_Sem_Pend(volatile struct RMP_Sem* Semaphore, rmp_ptr_t Slice)	
返回值	rmp_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。 该检查即使在 RMP_CHECK_ENABLE 关闭时也会执行。
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。	
	rmp_ptr_t Slice 暂无信号量时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_SLICE_MAX 之间的值，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_SLICE_MAX 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。	

2.3.11 等待信号量并解除调度器锁

本操作与等待信号量是类似的，其唯一区别是在调用它之前调度器必须不多不少地上一次锁。该调用无论成败，在返回时都会解除调度器锁，相当于“等待信号量”和“解除调度器锁”的原子化合并。这在实现复杂的客制化条件变量时很有用处。

表 2-25 等待信号量

函数原型	rmp_ret_t RMP_Sem_Pend_Unlock(volatile struct RMP_Sem* Semaphore, rmp_ptr_t Slice)	
返回值	rmp_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	调度器没有正确上锁，或者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。 最后两条检查即使在 RMP_CHECK_ENABLE 关闭时也会执行。
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。	
	rmp_ptr_t Slice 暂无信号量时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_SLICE_MAX 之间的值，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_SLICE_MAX 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。	

2.3.12 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，那么目标线程的等待信号量函数会返回 RMP_ERR_OPER。这个操作也同样可以用来解除消息队列与阻塞消息队列的等待，那些被解除等待的函数一样会返回 RMP_ERR_OPER。

表 2-26 解除信号量等待

函数原型	rmp_ret_t RMP_Sem_Abort(volatile struct RMP_Thd* Thread)	
返回值	rmp_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	该线程未处于等待信号量状态，无法被解除等待。 该检查即使在 RMP_CHECK_ENABLE 关闭时也会执行。

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。

2.3.13 查询信号量数目

本操作会查询信号量的当前数目。

表 2-27 查询信号量数目

函数原型	rmp_ret_t RMP_Sem_Cnt(volatile struct RMP_Sem* Semaphore)
返回值	rmp_ret_t 如果成功，返回当前信号量数目。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 指向要查询数目的信号量的信号量控制块的指针。

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。

在 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 为 0 时对应 127-64 Byte 的内存块，并且单个内存池的大小不能低于 1024 个机器字[1]。该分配器是为 128 MiB 以下的内存池特别优化的，不推荐将其用于更大的内存池。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 64 Byte。

内存池和线程的对应关系是非常复杂的，可以是一对一、一对多、多对一或多对多。为了最大化单线程场景下的性能，RMP 的内存管理接口不提供多线程安全性，这一点和其他内核接口非常不同。如果需要多线程安全性，可配合调度器锁或信号量自行实现。此外，不可在被分配的内存区域外进行读写，否则可能会破坏分配器的数据结构而导致异常。

所有的内存管理接口如下表所示：

表 2-28 内存管理接口一览

[1] 对于 16 位机不能少于 2kB，对于 32 位机则不能少于 4kB；若少于此数量，则无使用内存池的必要

接口	意义	特殊说明
RMP_Mem_Init	初始化内存池	内存池管理接口不提供任何线程安全性。多线程互斥操作必须由应用程序保证，这可以通过附加调度器锁或信号量解决。
RMP_Malloc	从内存池分配内存	
RMP_Free	向内存池归还内存	
RMP_Realloc	重新配置内存大小	

除 RMP_NULL 之外，它们可能返回的唯一返回值为 RMP_ERR_MEM（-8）。

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。

表 2-29 初始化内存池

函数原型	rmp_ret_t RMP_Mem_Init(volatile void* Pool, rmp_ptr_t Size)	
返回值	rmp_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 1024 个机器字。
参数	volatile void* Pool 指向要初始化为内存池的空白内存的指针。	
	rmp_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 64 Byte，如果试图分配的数量小于它而大于 0，则实际上会分配 64 Byte；如果传入的大小为 0 则会直接返回分配失败[1]。

表 2-30 从内存池分配内存

函数原型	void* RMP_Malloc(volatile void* Pool, rmp_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。

[1] 该检查在 RMP_CHECK_ENABLE 关闭时会被忽略

	rmp_ptr_t Size 要分配的内存块大小，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。Mem_Ptr 必须是由 RMP_Malloc 返回的某个尚未被归还的地址，并且必须归还到分配该内存时使用的内存池[1]，否则该操作会安静地失败。如果 Mem_Ptr 为 NULL，则不做任何操作。

表 2-31 向内存池归还内存

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。

2.4.4 重新配置内存大小

本操作会试图重新配置某内存块的大小。Mem_Ptr 必须是由 RMP_Malloc 返回的某个尚未被归还的地址，而且传入的内存池必须与调用 RMP_Malloc 时的内存池一致[2]。该函数在成功时，会返回原内存块或者一个前 Size 字节和原地址内容相同的新内存块，且在 Size 大于 0 而小于 64 时默认分配 64 Byte；若不成功，则返回 0（NULL）且原内存块不受影响。另外，如果 Size 的值为 0，该函数的行为与 RMP_Free 相同，会直接释放该内存块[3]；如果 Mem_Ptr 为 NULL，该函数的行为与 RMP_Malloc 相同，会直接分配新的内存块。

表 2-32 重新配置内存大小

函数原型	void* RMP_Realloc(volatile void* Pool, void* Mem_Ptr, rmp_ptr_t Size)
返回值	void* 如果成功，返回指向重新配置大小后的内存块的非 0（NULL）指针。如果失败则返回 0（NULL）。
参数	volatile void* Pool 指向要重新配置大小的内存块所在的内存池的指针。

[1] 这些检查在 RMP_CHECK_ENABLE 关闭时会被忽略

[2] 这些检查在 RMP_CHECK_ENABLE 关闭时会被忽略

[3] 这和某些 realloc 的实现可能不一致

	void* Mem_Ptr
	指向要重新配置大小的内存块的指针。
	rmp_ptr_t Size
	要将该内存块重新配置为的大小。该大小可以大于或者小于原内存块大小。

2.5 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。钩子函数可以分为必要钩子和可选钩子两类：前者被声明成函数且必须提供，后者则被声明成宏且可不填充。

2.5.1 必要钩子接口

必须由用户实现的钩子函数如下：

2.5.1.1 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。其典型用法是创建其它线程、信号量和消息队列，或者做一些系统初始化工作。需要注意的是，该钩子全程在关中断的条件下执行，因此不能调用任何可能依赖中断的操作，或者开中断的操作。如果需要调用这些操作，请在其它线程中执行。

表 2-33 空闲线程首次运行钩子

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

2.5.1.2 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。其典型用法是在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。不可以在该钩子中使空闲线程进入阻塞状态，否则系统会崩溃。

表 2-34 空闲线程反复运行钩子

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

2.5.2 可选钩子接口

可以由用户在需要时选择性实现的钩子宏如下：

2.5.2.1 系统启动钩子

该钩子宏若被定义，则会在系统完成处理器初始化后立即被调用。由于每个移植在系统启动时都仅仅会初始化一部分最基本的硬件，因此推荐将硬件的进一步初始化放置在这个钩子内部完成。如果所想要的基本系统配置与该平台中的配置头文件提供的基本初始化冲突[1]，那么也可以在该钩子内部重新完成系统基本配置。

表 2-35 系统启动钩子

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

2.5.2.2 系统上下文保存钩子

该钩子宏若被定义，则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[2]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。

表 2-36 系统上下文保存钩子

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

2.5.2.3 系统上下文恢复钩子

该钩子宏若被定义，则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[3]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。

表 2-37 系统上下文恢复钩子

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

2.5.2.4 调度器钩子

该钩子宏若被定义，则会在系统完成当前线程选择时被调用。如果需要实现无嘀嗒内核，可以在这个钩子内调用相关函数进行。具体的无嘀嗒系统实现方法请参见 7.1.1。

[1] 比如想要配置系统到完全不同的工作频率

[2] FPU 和其他外设寄存器组

[3] FPU 和其他外设寄存器组

表 2-38 调度器钩子

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

2.5.2.5 系统嘀嗒钩子

该钩子宏若被定义，则会在系统嘀嗒到来时被调用。

表 2-39 系统嘀嗒钩子

函数原型	void RMP_TIM_HOOK(rmp_ptr_t Slice)
返回值	无。
参数	rmp_ptr_t Slices _RMP_Tim_Handler 在被调用时接受的 Slice 参数。

2.5.2.6 延时插入钩子

该钩子宏若被定义，则会在有线程被插入延时队列时被调用。

表 2-40 延时插入钩子

函数原型	void RMP_DLY_HOOK(rmp_ptr_t Slice)
返回值	无。
参数	rmp_ptr_t Slices 带延时的系统函数被调用时接受的 Slice 参数。

2.5.2.7 错误记录钩子

该钩子宏若被定义，则会在系统打印错误信息时被调用。如果该钩子宏未被定义，则系统会默认将错误打印到控制台。

表 2-41 错误记录钩子

函数原型	void RMP_LOG(const char* File, long Line, const char* Date, const char* Time)
返回值	无。
参数	const char* File 发生错误的文件名。
	long Line

	发生错误的行号。
	const char* Date 编译日期。
	const char* Time 编译时间。

2.6 辅助系统接口

RMP 提供的其他辅助性接口包括了中断开关、调度器锁定解锁等实用功能。这些接口可以在用户编写应用程序的时候提供一些便利。

2.6.1 中断和调度器接口

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，因为在某些架构上 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，而且关中断的区间应当尽量短。

2.6.1.1 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。

表 2-42 除能中断

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

2.6.1.2 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。

表 2-43 使能中断

函数原型	void RMP_Int_Enable(void)
返回值	无。

参数	无。

2.6.1.3 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。

表 2-44 锁定调度器

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

2.6.1.4 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。

表 2-45 解锁调度器

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

2.6.2 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：

2.6.2.1 清零内存

本操作将一段内存清零。

表 2-46 清零内存

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

2.6.2.2 打印一个字符

本操作打印一个字符到控制台[1]。

表 2-47 打印一个字符

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。

2.6.2.3 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。

表 2-48 打印整形数字

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

2.6.2.4 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。

表 2-49 打印无符号整形数字

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

2.6.2.5 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。

表 2-50 打印字符串

[1] 一般是串口

函数原型	rmp_cnt_t RMP_Str_Print(rmp_s8_t* String)
返回值	rmp_cnt_t 返回打印的字符数量。
参数	rmp_s8_t* String 要打印的字符串。

2.6.2.6 得到一个字的最高位

本操作得到一个字的最高位的位号[1]。如果该数字为 0，那么需要返回-1[2]。这是一个通用函数，无法使用到处理器的特定加速指令；如果可能请尽量使用 RMP_MSB_GET 宏。

表 2-51 得到一个字的最高位

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

2.6.2.7 得到一个字的最低位

本操作得到一个字的最低位的位号[3]。如果该数字为 0，那么返回一个等于处理器字长的值[4]。这是一个通用函数，无法使用到处理器的特定加速指令；如果可能请尽量使用 RMP_LSB_GET 宏。

表 2-52 得到一个字的最低位

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

2.6.2.8 创建双向循环链表

本操作初始化双向循环链表的链表头。

---

[1] 比如 32 位处理器会返回 0-31

[2] 比如 32 位处理器应返回 0xFFFFFFFF

[3] 比如 32 位处理器会返回 0-31

[4] 比如 32 位处理器会返回 32

表 2-53 创建双向循环链表

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。

2.6.2.9 在双向循环链表中删除节点

本操作从双向链表中删除一个或一系列节点。

表 2-54 在双向链表中删除节点

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。

2.6.2.10 在双向循环链表中插入节点

本操作从双向链表中插入一个节点。

表 2-55 在双向循环链表中插入节点

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。

2.7 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 3 章　内核扩展接口

3.1　内核扩展接口简介

RMP 实现的基本内核接口原理清晰、易于验证、原子性强，但其过于基础，在应用程序开发中不便于直接使用。为解决此问题，RMP 提供一套易用性更好的内核扩展接口，它们均由一个或多个基本内核接口组合而来。由于这些接口并非原生，因而其速度较慢；在那些对性能有苛刻要求的场合，建议自行实现更适合应用场景的接口。

3.1.1　队列扩展接口

在消息队列方面，RMP 扩展先进先出队列（First-In-First-Out，FIFO）、消息队列（Message Queue）和阻塞消息队列（Blocking Message Queue）。

先进先出队列类似于 Unix 系统中的非阻塞管道。它们不支持阻塞，仅仅是链表的一个线程安全封装，可用来以轮询方式传递数据。它的速度最快。

消息队列类似于其它各类操作系统中的同名机制。它可以支持多个生产者和多个消费者，其中只有消费者端具备阻塞功能，生产者端则不具备阻塞功能，且队列的长度不受限制。它是计数信号量和先进先出队列的联合封装，其速度中等。

阻塞消息队列在消息队列的基础上支持队列长度限制与生产者端阻塞，可在消费者无法吃下全部数据时让生产者放慢速度。它是两个计数信号量和先进先出队列的联合封装，其功能最强大，但资源消耗最大、速度最慢。

队列机制的消息封装均采用双向链表头结构体 struct RMP_List。如果需要传递复杂的消息，可以自定义包含该结构体的消息结构体，并将 struct RMP_List 作为它的第一个元素，这样就可以通过双向链表头结构体方便地找到消息结构体。考虑到灵活性，结构体的空间分配和提供均由调用者完成，通信机制本身并不负责这些空间的分配和管理；这有助于避免无谓的空间浪费和数据拷贝。虽然队列是线程安全的，但所有的消息都是线程局部（Thread-Local）和队列局部（Queue-Local）的[1]；禁止让两个线程同时操作一条消息，也禁止将同一消息插入两个队列；一旦消息入队，即禁止对其进行修改。

消息队列扩展接口不是原子接口，而是由多个原子接口组装而来。若在线程正使用消息队列时删除线程可能留下不完整的队列数据结构；若在消息队列正被使用时删除它然后立即利用同样的控制块创建新队列则可能引发 ABA 问题。因此，若需要删除线程或消息队列，请保证上述情况不存在，或者按照如下的顺序执行强制删除：先（1）删除所有使用该消息队列的线程，然后（2）删除该消息队列，最后（3）如有需要再重新创建之。值得注意的是，在（1）中，部分线程可能刚从内存池中分配到消息，还没来得及将其挂接到队列上即遭删除。此时，回收泄露的消息是应用程序的责任；这可以通过内存块标记法实现，也可以给消息配备专用的内存池并重新初始化该内存池。此外，在线程使用消息队列时可随时将其悬起，或者使用 RMP_Sem_Abort 解除其在队列上的阻塞状态。

[1] 对每条消息加锁以防止竞争冒险会损失太多性能，得不偿失

3.1.2 软定时器扩展接口

在定时回调方面，RMP 扩展了软定时器（Alarm）接口。通过该接口，应用程序可以注册一个在一段时间之后才被调用的回调函数以实现定时触发功能。为了（1）提高定时器功能的灵活性和（2）防止高优先级任务的定时功能被低优先级任务干扰，RMP 提供多个定时器管理器（Manager）：每个定时器管理器运行在何优先级及其采用何定时粒度可以被任意定制。

因此，RMP 的软定时器接口被分成了（1）定时器管理器操作和（2）定时器操作接口。要创建定时器，应用程序需要先（1）创建定时器管理器并定期维护之，然后（2）初始化所需的定时器，最后（3）将定时器注册到相应的定时器管理器。一旦定时器超时，该定时器对应的回调钩子函数（Callback Hook）就会被调用；在调用该钩子时，当前定时器管理器 Amgr、当前定时器 Alrm 以及超期程度 Overdue 会作为参数传入以方便应用程序设计。同一定时器管理器上注册的定时器总是先按照（1）溢出时间，再按照（2）注册顺序触发：如果一个定时器的溢出时间比另一个早，则溢出时间早的定时器先触发；如果两个定时器同时溢出，则先注册的定时器先触发。

软定时器机制的定时器管理器封装均采用 struct RMP_Amgr，定时器封装均采用 struct RMP_Alrm。如果需要在这些数据结构上附加数据，可以自定义包含上述结构体的新结构体，并将 struct RMP_Amgr 或 struct RMP_Alrm 作为它的第一个元素，这样就可以通过上述结构体方便地找到附加数据。考虑到灵活性，上述结构体的空间分配和提供均由调用者完成，软定时器机制本身并不负责这些空间的分配和管理；这有助于避免无谓的空间浪费和数据拷贝。虽然定时器管理器是线程安全的，但所有的定时器都是线程局部（Thread-Local）和定时器管理器局部（Manager-Local）的[1]；禁止让两个线程同时操作一个定时器[2]，也禁止将同一定时器注册到两个定时器管理器[3]；一旦定时器被注册，即禁止对其进行重新初始化。

软定时器扩展接口不是原子接口，而是由多个原子接口组装而来。若在线程正使用定时器接口时删除线程可能留下不完整的定时器管理器数据结构；若在定时器管理器正被使用时删除它然后立即利用同样的控制块创建新定时器管理器则可能引发 ABA 问题。因此，若需要删除线程或定时器管理器，请保证上述情况不存在，或者按照如下的顺序执行强制删除：先（1）删除所有操作该定时器管理器的线程，然后（2）删除该定时器管理器，最后（3）如有需要再重新创建之。值得注意的是，在（1）中，部分线程可能刚从内存池中分配到定时器，还没来得及将其挂接到队列上即遭删除。此时，回收泄露的定时器是应用程序的责任；这可以通过内存块标记法实现，也可以给定时器配备专用的内存池并重新初始化该内存池。此外，在线程使用定时器时不推荐将其悬起，这是由于定时器管理器操作是使用信号量[4]而非调度器锁保护的。如果线程在持有信号量时被悬起，则其它使用同一定时器管理器的线程可能会被无限阻塞直到线程悬起被解除。

---

[1] 对每个定时器加锁以防止竞争冒险会损失太多性能，得不偿失

[2] 例外情况：如果两个线程同时操作一个定时器，但它们的操作都指定了相同的定时器管理器，则此种操作是允许的，因为定时器管理器的信号量会保证互斥访问

[3] 接口会在非并发条件下探测到此种错误并进行报告，但在并发条件下则不保证能探测到此种错误

[4] 在这里作为互斥锁使用

3.2 先进先出队列扩展接口

所有的先进先出队列扩展接口如下表所示：

表 3-1 先进先出队列扩展接口一览

接口	意义	锁调度器时调用	中断中调用
RMP_Fifo_Crt	创建先进先出队列	可	不可
RMP_Fifo_Del	删除先进先出队列	可	不可
RMP_Fifo_Read	读先进先出队列	可	不可
RMP_Fifo_Write	写先进先出队列	可	不可
RMP_Fifo_Write_ISR	从中断写先进先出队列	不可	可
RMP_Fifo_Cnt	查询先进先出队列长度	可	不可

它们可能有如下返回值：

表 3-2 先进先出队列扩展接口的可能返回值

返回值	数值	意义
RMP_ERR_OPER	-6	由于其他原因导致操作失败。
RMP_ERR_FIFO	-9	由于先进先出队列控制块相关的原因导致操作失败。

3.2.1 创建先进先出队列

本操作会创建一个先进先出队列。

表 3-3 创建先进先出队列

函数原型	rmp_ret_t RMP_Fifo_Crt(volatile struct RMP_Fifo* Fifo)
返回值	rmp_ret_t 如果成功，返回 0。如果失败则可能有如下返回值： RMP_ERR_FIFO 先进先出队列控制块为 0（NULL）或者已被使用。
参数	volatile struct RMP_Fifo* Fifo 指向空的、将被用于该先进先出队列的先进先出队列控制块的指针。

3.2.2 删除先进先出队列

本操作会创建一个先进先出队列。只有空的先进先出队列才能被删除。

表 3-4 删除先进先出队列

函数原型	rmp_ret_t RMP_Fifo_Del(volatile struct RMP_Fifo* Fifo)	
返回值	rmp_ret_t 如果成功，返回 0。如果失败则可能有如下返回值：	
	RMP_ERR_FIFO	先进先出队列控制块为 0（NULL）或者未被使用。
	RMP_ERR_OPER	先进先出队列中还有元素，无法删除。 该检查即使在 RMP_CHECK_ENABLE 关闭时也会执行。
参数	volatile struct RMP_Fifo* Fifo 指向要删除的先进先出队列的先进先出队列控制块的指针。	

3.2.3 读先进先出队列

本操作会试图从先进先出队列中读取一个元素。如果先进先出队列中无元素，本操作会即刻返回，不提供阻塞功能。

表 3-5 读先进先出队列

函数原型	rmp_ret_t RMP_Fifo_Read(volatile struct RMP_Fifo* Fifo, volatile struct RMP_List** Node)	
返回值	rmp_ret_t 如果成功，返回 0。如果失败则可能有如下返回值：	
	RMP_ERR_FIFO	先进先出队列控制块为 0（NULL）或者未被使用。
	RMP_ERR_OPER	读指针 Node 为 0（NULL），或者先进先出队列中暂时没有元素。 最后一条检查即使在 RMP_CHECK_ENABLE 关闭时也会执行。
参数	volatile struct RMP_Fifo* Fifo 指向要读的先进先出队列的先进先出队列控制块的指针。	
	volatile struct RMP_List** Node 该参数用于输出，输出读取的元素。	

3.2.4 写先进先出队列

本操作会向先进先出队列中写入一个元素。

表 3-6 写先进先出队列

函数原型	rmp_ret_t RMP_Fifo_Write(volatile struct RMP_Fifo* Fifo, volatile struct RMP_List* Node)
返回值	rmp_ret_t

	如果成功，返回 0。如果失败则可能有如下返回值：
	RMP_ERR_FIFO 先进先出队列控制块为 0（NULL）或者未被使用。
	RMP_ERR_OPER 写指针 Node 为 0（NULL）。
参数	volatile struct RMP_Fifo* Fifo 指向要写的先进先出队列的先进先出队列控制块的指针。
	volatile struct RMP_List* Node 指向要写入的元素的指针。

3.2.5 从中断写先进先出队列

本操作会从中断向量中向先进先出队列中写入一个元素。

表 3-7 从中断写先进先出队列

函数原型	rmp_ret_t RMP_Fifo_Write(volatile struct RMP_Fifo* Fifo, volatile struct RMP_List* Node)
返回值	rmp_ret_t 如果成功，返回 0。如果失败则可能有如下返回值：
	RMP_ERR_FIFO 先进先出队列控制块为 0（NULL）或者未被使用。
	RMP_ERR_OPER 写指针 Node 为 0（NULL）。
参数	volatile struct RMP_Fifo* Fifo 指向要写的先进先出队列的先进先出队列控制块的指针。
	volatile struct RMP_List* Node 指向要写入的元素的指针。

3.2.6 查询先进先出队列长度

本操作会查询先进先出队列的当前长度。

表 3-8 查询先进先出队列长度

函数原型	rmp_ret_t RMP_Fifo_Cnt(volatile struct RMP_Fifo* Fifo)
返回值	rmp_ret_t 如果成功，返回当前队列长度。如果失败则可能有如下返回值：
	RMP_ERR_FIFO 先进先出队列控制块为 0（NULL）或者未被使用。
参数	volatile struct RMP_Fifo* Fifo 指向要查询长度的先进先出队列的先进先出队列控制块的指针。

3.3 消息队列扩展接口

所有的消息队列扩展接口如下表所示：

表 3-9 消息队列扩展接口一览

接口	意义	锁调度器时调用	中断中调用
RMP_Msgq_Crt	创建消息队列	可	不可
RMP_Msgq_Del	删除消息队列	可	不可
RMP_Msgq_Snd	向消息队列发送	可	不可
RMP_Msgq_Snd_ISR	从中断向消息队列发送	不可	可
RMP_Msgq_Rcv	从消息队列接收	不可	不可
RMP_Msgq_Cnt	查询消息队列长度	可	不可

它们可能有如下返回值：

表 3-10 消息队列扩展接口的可能返回值

返回值	数值	意义
RMP_ERR_OPER	-6	由于其他原因导致操作失败。
RMP_ERR_MSGQ	-10	由于消息队列控制块相关的原因导致操作失败。

3.3.1 创建消息队列

本操作会创建一个消息队列。

表 3-11 创建消息队列

函数原型	rmp_ret_t RMP_Msgq_Crt(volatile struct RMP_Msgq* Queue)	
返回值	rmp_ret_t 如果成功，返回 0。如果失败则可能有如下返回值：	
	RMP_ERR_MSGQ	消息队列控制块为 0（NULL）或者已被使用。
参数	volatile struct RMP_Msgq* Queue 指向空的、将被用于该消息队列的消息队列控制块的指针。	

3.3.2 删除消息队列

本操作会删除一个消息队列。只有空的消息队列才能被删除。

表 3-12 删除消息队列

函数原型	rmp_ret_t RMP_Msgq_Del(volatile struct RMP_Msgq* Queue)	
返回值	rmp_ret_t 如果成功，返回 0。如果失败则可能有如下返回值：	
	RMP_ERR_MSGQ	消息队列控制块为 0（NULL）或者未被使用。
	RMP_ERR_OPER	消息队列中还有元素，无法删除。 该检查即使在 RMP_CHECK_ENABLE 关闭时也会执行。
参数	volatile struct RMP_Msgq* Queue 指向要删除的消息队列的消息队列控制块的指针。	

3.3.3 向消息队列发送

本操作会向消息队列中发送一条消息。

表 3-13 向消息队列发送

函数原型	rmp_ret_t RMP_Msgq_Snd(volatile struct RMP_Msgq* Queue, volatile struct RMP_List* Node)	
返回值	rmp_ret_t 如果成功，返回 0。如果失败则可能有如下返回值：	
	RMP_ERR_MSGQ	消息队列控制块为 0（NULL）或者未被使用。
	RMP_ERR_OPER	发送指针 Node 为 0（NULL），或者队列中当前的消息数量达到 RMP_SEM_CNT_MAX。
参数	volatile struct RMP_Msgq* Queue 指向要发送到的消息队列的消息队列控制块的指针。	
	volatile struct RMP_List* Node 指向要发送的消息的指针。	

3.3.4 从中断向消息队列发送

本操作会在中断中向消息队列中发送一条消息。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。

表 3-14 从中断向消息队列发送

函数原型	rmp_ret_t RMP_Msgq_Snd_ISR(volatile struct RMP_Msgq* Queue,

	volatile struct RMP_List* Node)
返回值	rmp_ret_t 如果成功，返回 0。如果失败则可能有如下返回值：
	RMP_ERR_MSGQ 消息队列控制块为 0（NULL）或者未被使用。
	RMP_ERR_OPER 发送指针 Node 为 0（NULL），或者队列中当前的消息数量达到 RMP_SEM_CNT_MAX。
参数	volatile struct RMP_Msgq* Queue 指向要发送到的消息队列的消息队列控制块的指针。
	volatile struct RMP_List* Node 指向要发送的消息的指针。

3.3.5 从消息队列接收

本操作会从消息队列中接收一条消息。

表 3-15 从消息队列接收

函数原型	rmp_ret_t RMP_Msgq_Rcv(volatile struct RMP_Msgq* Queue, volatile struct RMP_List** Node, rmp_ptr_t Slice)
返回值	rmp_ret_t 如果成功，返回 0。如果失败则可能有如下返回值：
	RMP_ERR_MSGQ 消息队列控制块为 0（NULL）或者未被使用。
	RMP_ERR_OPER 接收指针 Node 为 0（NULL），或者在不阻塞条件下探测到可能造成阻塞，或者等待因超时、目标队列被删除、被中途解除而失败。最后两条检查即使在 RMP_CHECK_ENABLE 关闭时也会执行。
参数	volatile struct RMP_Msgq* Queue 指向要接收消息的消息队列的消息队列控制块的指针。
	volatile struct RMP_List** Node 该参数用于输出，输出读取的消息。
	rmp_ptr_t Slice 暂无消息时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_SLICE_MAX 之间的值，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_SLICE_MAX 的值，那么意味着将永远阻塞直到接收到消息或者消息队列被销毁。

3.3.6 查询消息队列长度

本操作会查询消息队列的当前长度。

表 3-16 查询消息队列长度

函数原型	rmp_ret_t RMP_Msgq_Cnt(volatile struct RMP_Msgq* Queue)
返回值	rmp_ret_t 如果成功，返回当前队列长度。如果失败则可能有如下返回值： RMP_ERR_MSGQ 消息队列控制块为 0（NULL）或者未被使用。
参数	volatile struct RMP_Msgq* Queue 指向要查询长度的消息队列的消息队列控制块的指针。

3.4 阻塞消息队列扩展接口

所有的阻塞消息队列扩展接口如下表所示：

表 3-17 阻塞消息队列扩展接口一览

接口	意义	锁调度器时调用	中断中调用
RMP_Bmq_Crt	创建阻塞消息队列	可	不可
RMP_Bmq_Del	删除阻塞消息队列	可	不可
RMP_Bmq_Snd	向阻塞消息队列发送	不可	不可
RMP_Bmq_Snd_ISR	从中断向阻塞消息队列发送	不可	可
RMP_Bmq_Rcv	从阻塞消息队列接收	不可	不可
RMP_Bmq_Cnt	查询阻塞消息队列长度	可	不可

它们可能有如下返回值：

表 3-18 阻塞消息队列扩展接口的可能返回值

返回值	数值	意义
RMP_ERR_OPER	-6	由于其他原因导致操作失败。
RMP_ERR_BMQ	-11	由于阻塞消息队列控制块相关的原因导致操作失败。

3.4.1 创建阻塞消息队列

本操作会创建一个阻塞消息队列。

表 3-19 创建阻塞消息队列

函数原型	rmp_ret_t RMP_Bmq_Crt(volatile struct RMP_Bmq* Queue, rmp_ptr_t Limit)	
返回值	rmp_ret_t 如果成功，返回 0。如果失败则可能有如下返回值：	
	RMP_ERR_BMQ	阻塞消息队列控制块为 0（NULL）或者已被使用。
	RMP_ERR_OPER	传入的最大队列长度为 0，或者达到了 RMP_SEM_CNT_MAX。
参数	volatile struct RMP_Bmq* Queue 指向空的、将被用于该阻塞消息队列的阻塞消息队列控制块的指针。	
	rmp_ptr_t Limit 最大队列长度：队列中最多允许同时存在这个数量的消息。	

3.4.2 删除阻塞消息队列

本操作会删除一个阻塞消息队列。只有空的阻塞消息队列才能被删除。

表 3-20 删除阻塞消息队列

函数原型	rmp_ret_t RMP_Bmq_Del(volatile struct RMP_Bmq* Queue)	
返回值	rmp_ret_t 如果成功，返回 0。如果失败则可能有如下返回值：	
	RMP_ERR_BMQ	阻塞消息队列控制块为 0（NULL）或者未被使用。
	RMP_ERR_OPER	阻塞消息队列中还有元素，无法删除。 该检查即使在 RMP_CHECK_ENABLE 关闭时也会执行。
参数	volatile struct RMP_Bmq* Queue 指向要删除的阻塞消息队列的阻塞消息队列控制块的指针。	

3.4.3 向阻塞消息队列发送

本操作会向阻塞消息队列中发送一条消息。

表 3-21 向阻塞消息队列发送

函数原型	rmp_ret_t RMP_Bmq_Snd(volatile struct RMP_Bmq* Queue, volatile struct RMP_List* Node, rmp_ptr_t Slice)
返回值	rmp_ret_t 如果成功，返回 0。如果失败则可能有如下返回值：

	RMP_ERR_BMQ	阻塞消息队列控制块为 0（NULL）或者未被使用。
	RMP_ERR_OPER	发送指针 Node 为 0（NULL），或者队列中当前的消息数量达到 RMP_SEM_CNT_MAX，或者在不阻塞条件下探测到可能造成阻塞，或者等待因超时、目标队列被删除、被中途解除而失败。 最后两条检查即使在 RMP_CHECK_ENABLE 关闭时也会执行。
参数	volatile struct RMP_Bmq* Queue 指向要发送到的消息队列的消息队列控制块的指针。	
	volatile struct RMP_List* Node 指向要发送的消息的指针。	
	rmp_ptr_t Slice 队列暂满时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_SLICE_MAX 之间的值，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_SLICE_MAX 的值，那么意味着将永远阻塞直到发送完消息。	

3.4.4 从中断向阻塞消息队列发送

本操作会在中断中向阻塞消息队列中发送一个元素。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，而且也不接受等待时间片参数。

表 3-22 从中断向阻塞消息队列发送

函数原型	rmp_ret_t RMP_Bmq_Snd_ISR(volatile struct RMP_Bmq* Queue, volatile struct RMP_List* Node)	
返回值	rmp_ret_t 如果成功，返回 0。如果失败则可能有如下返回值：	
	RMP_ERR_BMQ	阻塞消息队列控制块为 0（NULL）或者未被使用。
	RMP_ERR_OPER	发送指针 Node 为 0（NULL），或者队列中当前的消息数量达到 RMP_SEM_CNT_MAX，或者队列中的消息数目已经达到创建时允许的最大数目。 最后一条检查即使在 RMP_CHECK_ENABLE 关闭时也会执行。
参数	volatile struct RMP_Bmq* Queue 指向要发送到的阻塞消息队列的阻塞消息队列控制块的指针。	
	volatile struct RMP_List* Node 指向要发送的消息的指针。	

3.4.5 从阻塞消息队列接收

本操作会从阻塞消息队列中接收一条消息。

表 3-23 从阻塞消息队列接收

函数原型	rmp_ret_t RMP_Bmq_Rcv(volatile struct RMP_Bmq* Queue, volatile struct RMP_List** Node, rmp_ptr_t Slice)	
返回值	rmp_ret_t 如果成功，返回 0。如果失败则可能有如下返回值：	
	RMP_ERR_BMQ	阻塞消息队列控制块为 0（NULL）或者未被使用。
	RMP_ERR_OPER	接收指针 Node 为 0（NULL），或者在不阻塞条件下探测到可能造成阻塞，或者等待因超时、目标队列被删除、被中途解除而失败。最后两条检查即使在 RMP_CHECK_ENABLE 关闭时也会执行。
参数	volatile struct RMP_Bmq* Queue 指向要接收消息的阻塞消息队列的阻塞消息队列控制块的指针。	
	volatile struct RMP_List** Node 该参数用于输出，输出接收的消息。	
	rmp_ptr_t Slice 暂无消息时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_SLICE_MAX 之间的值，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_SLICE_MAX 的值，那么意味着将永远阻塞直到接收到消息或者阻塞消息队列被销毁。	

3.4.6 查询阻塞消息队列长度

本操作会查询阻塞消息队列的当前长度。

表 3-24 查询阻塞消息队列长度

函数原型	rmp_ret_t RMP_Bmq_Cnt(volatile struct RMP_Bmq* Queue)	
返回值	rmp_ret_t 如果成功，返回当前队列长度。如果失败则可能有如下返回值：	
	RMP_ERR_BMQ	阻塞消息队列控制块为 0（NULL）或者未被使用。
参数	volatile struct RMP_Bmq* Queue 指向要查询长度的阻塞消息队列的阻塞消息队列控制块的指针。	

3.5 软定时器扩展接口

所有的软定时器扩展接口如下表所示：

表 3-25 软定时器扩展接口一览

接口	意义	锁调度器时调用	中断中调用
RMP_Amgr_Crt	创建定时器管理器	可	不可
RMP_Amgr_Del	删除定时器管理器	可	不可
RMP_Amgr_Pop	拆解定时器管理器	可	不可
RMP_Amgr_Proc	维护定时器管理器	不可	不可
RMP_Amgr_Cnt	查询定时器数量	可	不可
RMP_Alrm_Init	初始化定时器	可	不可
RMP_Alrm_Set	注册定时器	不可	不可
RMP_Alrm_Clr	取消定时器	不可	不可
RMP_Alrm_Trg	触发定时器	不可	不可

它们可能有如下返回值：

表 3-26 软定时器扩展接口的可能返回值

返回值	数值	意义
RMP_ERR_OPER	-6	由于其他原因导致操作失败。
RMP_ERR_AMGR	-12	由于定时器管理器控制块相关的原因导致操作失败。
RMP_ERR_ALRM	-13	由于定时器控制块相关的原因导致操作失败。

3.5.1 创建定时器管理器

本操作会创建一个定时器管理器。

表 3-27 创建定时器管理器

函数原型	rmp_ret_t RMP_Amgr_Crt(volatile struct RMP_Amgr* Amgr)
返回值	rmp_ret_t 如果成功，返回 0。如果失败则可能有如下返回值： RMP_ERR_AMGR 定时器管理器控制块为 0（NULL）或者已被使用。
参数	volatile struct RMP_Amgr* Amgr 指向空的、将被用于该定时器管理器的定时器管理器控制块的指针。

3.5.2 删除定时器管理器

本操作会删除一个定时器管理器。只有空的定时器管理器才能被删除。如果决定删除定时器管理器，要么等待（1）其中的定时器自然超时，要么（2）使用下一小节所述的拆解函数手动拆解之。

表 3-28 删除定时器管理器

函数原型	rmp_ret_t RMP_Amgr_Del(volatile struct RMP_Amgr* Amgr)	
返回值	rmp_ret_t 如果成功，返回 0。如果失败则可能有如下返回值：	
	RMP_ERR_AMGR	定时器管理器控制块为 0（NULL）或者未被使用。
	RMP_ERR_OPER	定时器管理器中还有定时器，无法删除。 该检查即使在 RMP_CHECK_ENABLE 关闭时也会执行。
参数	volatile struct RMP_Amgr* Amgr 指向要删除的定时器管理器的定时器管理器控制块的指针。	

3.5.3 拆解定时器管理器

本操作会从定时器管理器中强行弹出一个定时器。在需要强行停止定时器管理器运作的场合，在终止一切可能操作该定时器管理器的线程后，定时器管理器的数据结构可能遭到破坏。此时，使用本函数可以对定时器管理器进行逐步拆解，使其中不再含有任何定时器，以便对其进行删除。如果定时器是从内存池中分配的，可以将它们一一退还给内存池。

在拆解定时器之前，一切定时器管理器操作都必须停止；一旦开始拆解任何定时器，后续就只允许对它（1）进行拆解或（2）进行删除。

表 3-29 拆解定时器管理器

函数原型	rmp_ret_t RMP_Amgr_Pop(volatile struct RMP_Amgr* Amgr, volatile struct RMP_Alrm** Alrm)	
返回值	rmp_ret_t 如果成功，返回 0。如果失败则可能有如下返回值：	
	RMP_ERR_AMGR	定时器管理器控制块为 0（NULL）或者未被使用。
	RMP_ERR_OPER	读指针 Alrm 为 0（NULL），或者定时器管理器中已无定时器。 最后一条检查即使在 RMP_CHECK_ENABLE 关闭时也会执行。
参数	volatile struct RMP_Amgr* Amgr 指向要拆解的定时器管理器的定时器管理器控制块的指针。	
	volatile struct RMP_Alrm** Alrm 该参数用于输出，输出拆解出的定时器。	

3.5.4 维护定时器管理器

本操作会对定时器管理器增加时间计数，并处理任何超时的定时器，是定时器管理器的核心接口。本操作需要被定期调用以处理注册于管理器上的定时器；可以安排一个专用线程来周期性地调用它，调用周期即是该定时器管理器的计时粒度。

表 3-30 维护定时器管理器

函数原型	rmp_ret_t RMP_Amgr_Proc(volatile struct RMP_Amgr* Amgr, rmp_ptr_t Tick)	
返回值	rmp_ret_t 如果成功，返回定时器管理器上的当前定时器数量。如果失败则可能有如下返回值：	
	RMP_ERR_AMGR	定时器管理器控制块为 0（NULL）或者未被使用。
	RMP_ERR_OPER	Tick 为 0 或超过 RMP_MASK_INTMAX，或者加锁操作因定时器管理器被删除、被取消而失败。 最后一条检查即使在 RMP_CHECK_ENABLE 关闭时也会执行。
参数	volatile struct RMP_Amgr* Amgr 指向要维护的定时器管理器的定时器管理器控制块的指针。	
	rmp_ptr_t Tick 要给该定时器管理器增加的时间计数。这个数字不受 RMP_SLICE_MAX 限制，但它不能为 0，也不能超过 2^{N-1}-1（也即 RMP_MASK_INTMAX），其中 N 是 RMP_WORD_ORDER。一次性增加多于一个时间计数可能导致定时器超期溢出，此时其回调钩子函数被调用时其超期程度 Overdue 会是一个正数。	

3.5.5 查询定时器数量

本操作会查询定时器管理器上的当前定时器数量。

表 3-31 查询定时器数量

函数原型	rmp_ret_t RMP_Amgr_Cnt(volatile struct RMP_Amgr* Amgr)	
返回值	rmp_ret_t 如果成功，返回定时器管理器上的当前定时器数量。如果失败则可能有如下返回值：	
	RMP_ERR_AMGR	定时器管理器控制块为 0（NULL）或者未被使用。
参数	volatile struct RMP_Amgr* Amgr 指向要查询定时器数量的定时器管理器的定时器管理器控制块的指针。	

3.5.6 初始化定时器

本操作会依据用户给出的参数初始化定时器的数据结构，以便接下来将其注册到定时器管理器。

表 3-32 初始化定时器

函数原型	rmp_ret_t RMP_Alrm_Init(volatile struct RMP_Alrm* Alrm, rmp_ptr_t Delay, rmp_ptr_t Mode, void (*Hook)(volatile struct RMP_Amgr*, volatile struct RMP_Alrm*, rmp_cnt_t))
返回值	rmp_ret_t 如果成功，返回 0。如果失败则可能有如下返回值： RMP_ERR_ALRM 定时器控制块为 0（NULL）或者已被注册。 RMP_ERR_OPER Delay 为 0 或超过 RMP_MASK_INTMAX，或者 Mode 既不是 RMP_ALRM_AUTORLD 也不是 RMP_ALRM_ONESHOT，或者 Hook 为 0（NULL）。
参数	volatile struct RMP_Alrm* Alrm 指向要初始化的定时器的定时器控制块的指针。 rmp_ptr_t Delay 该定时器的延时时间数。这个数字不受 RMP_SLICE_MAX 限制，但它不能为 0，也不能超过 2^{N-1}-1（也即 RMP_MASK_INTMAX），其中 N 是 RMP_WORD_ORDER。该数字的单位即是定时器管理器的时间粒度。 rmp_ptr_t Mode 该定时器的工作模式，可以是自动重装（RMP_ALRM_AUTORLD）或者单次触发（RMP_ALRM_ONESHOT）。 void (*Hook)(volatile struct RMP_Amgr*, volatile struct RMP_Alrm*, rmp_cnt_t) 该定时器的回调钩子函数。其具体原型如下表所示。

表 3-33 定时器回调函数钩子

函数原型	void Hook(volatile struct RMP_Amgr* Amgr, volatile struct RMP_Alrm* Alrm, rmp_cnt_t Overdue)
返回值	无。
参数	volatile struct RMP_Amgr* Amgr

	指向该定时器所属的定时器管理器的定时器管理器结构体的指针。
	volatile struct RMP_Alrm* Alrm 指向该定时器的定时器结构体的指针。
	rmp_cnt_t Overdue 该定时器的超期程度。如果该定时器因 RMP_Amgr_Proc 一次增加多个时间计数而超期溢出，则该数是表征超期程度的正数；如果该定时器因 RMP_Alrm_Trig 而被提早触发，则该数是表征提早程度的负数；如果该定时器正常溢出，则该数为 0。

3.5.7 注册定时器

本操作会将指定的定时器注册到指定的定时器管理器。如果定时器已被注册到同一个定时器管理器，则其延时时间将被重置。

表 3-34 注册定时器

函数原型	rmp_ret_t RMP_Alrm_Set(volatile struct RMP_Amgr* Amgr, volatile struct RMP_Alrm* Alrm)	
返回值	rmp_ret_t 如果成功，返回 0。如果失败则可能有如下返回值：	
	RMP_ERR_AMGR	定时器管理器控制块为 0（NULL）或者未被使用。
	RMP_ERR_ALRM	定时器控制块为 0（NULL），或者被注册到了其它定时器管理器。 最后一条检查即使在 RMP_CHECK_ENABLE 关闭时也会执行。
	RMP_ERR_OPER	加锁操作因定时器管理器被删除、被取消而失败。 该检查即使在 RMP_CHECK_ENABLE 关闭时也会执行。
参数	volatile struct RMP_Amgr* Amgr 指向要注册到的定时器管理器的定时器管理器结构体的指针。	
	volatile struct RMP_Alrm* Alrm 指向要注册的定时器的定时器控制块的指针。	

3.5.8 取消定时器

本操作会取消指定的定时器。被取消的定时器将被立即从定时器管理器解除注册。

表 3-35 提前触发定时器

函数原型	rmp_ret_t RMP_Alrm_Clr(volatile struct RMP_Amgr* Amgr, volatile struct RMP_Alrm* Alrm)
返回值	rmp_ret_t

	如果成功，返回 0。如果失败则可能有如下返回值：	
	RMP_ERR_AMGR	定时器控制器控制块为 0（NULL）或者未被使用。
	RMP_ERR_ALRM	定时器控制块为 0（NULL），或者未被注册到该定时器管理器。 最后一条检查即使在 RMP_CHECK_ENABLE 关闭时也会执行。
	RMP_ERR_OPER	加锁操作因定时器管理器被删除、被取消而失败。 该检查即使在 RMP_CHECK_ENABLE 关闭时也会执行。
参数	volatile struct RMP_Amgr* Amgr 指向该定时器所在的定时器管理器的定时器管理器结构体的指针。	
	volatile struct RMP_Alrm* Alrm 指向要取消的定时器的定时器控制块的指针。	

3.5.9 触发定时器

本操作会提前触发指定的定时器，好像其定时时间已到那样。定时器的回调钩子函数会被调用，且其超期程度 Overdue 将是一个负值。在此之后，自动重装的定时器会立即再次开始计时，而单次触发的定时器则会从其定时器管理器解除注册。

表 3-36 触发定时器

函数原型	rmp_ret_t RMP_Alrm_Trg(volatile struct RMP_Amgr* Amgr, volatile struct RMP_Alrm* Alrm)	
返回值	rmp_ret_t 如果成功，返回 0。如果失败则可能有如下返回值：	
	RMP_ERR_AMGR	定时器管理器控制块为 0（NULL）或者未被使用。
	RMP_ERR_ALRM	定时器控制块为 0（NULL），或者未被注册到该定时器管理器。 最后两条检查即使在 RMP_CHECK_ENABLE 关闭时也会执行。
	RMP_ERR_OPER	加锁操作因定时器管理器被删除、被取消而失败。 该检查即使在 RMP_CHECK_ENABLE 关闭时也会执行。
参数	volatile struct RMP_Amgr* Amgr 指向要提前触发的定时器所在的定时器管理器的定时器管理器结构体的指针。	
	volatile struct RMP_Alrm* Alrm 指向要提前触发的定时器的定时器控制块的指针。	

3.6 本章参考文献

无

第 4 章　轻量级图形界面接口

4.1　嵌入式图形界面简介

嵌入式图形界面（Graphic User Interface，GUI）是用于嵌入式系统的专用图形解决方案。和传统的计算机图形界面相比，嵌入式 GUI 具有简单性、多样性、平面性和受限性的特点。抓住这四点特征是良好完成嵌入式 GUI 设计的关键。

4.1.1　简单性

通常而言，嵌入式 GUI 的组成元素比较单一，一般是几个按钮或者几个显示控件，不像传统 GUI 一样有滚动条、树状列表等诸多组成元素。而且，一旦 UI 界面设计完成，在产品使用过程中就很少改变。

4.1.2　多样性

传统 GUI 的设计在经过多年演化后，基本上达成了一定的共识，形成了如 Windows，MacOS，Gnome 和 Unity 为代表的一大批桌面环境。它们都强调优化键鼠操作的使用体验，并使用近乎相同的使用逻辑，其控件大多标准化。而在嵌入式 GUI 中则正好相反，很少使用标准键鼠输入而较多使用定制键盘和触摸屏，并且有大量的非标准控件和界面。这些控件的界面和逻辑也是趣旨各异的，需要针对具体应用做大量的优化和适配。

4.1.3　平面性

传统 GUI 的技术栈非常强调采用复杂的窗口裁剪，使得窗口具备垂直向的叠加关系，并且需要通过复杂的算法计算出各个窗口的显示区域，并且予以绘制。这在多窗口多任务的极度复杂环境中是必须的。而在嵌入式 GUI 中，由于显示区域本来就很小，再加上输入手段受限，很少用到多个窗口之间的纵向空间关系，而更加强调扁平而人性化的 GUI 设计。这一点从各个手机操作系统的演化上也可以看出端倪。

4.1.4　受限性

传统 GUI 的实现往往有赖于强有力的 CPU 和 GPU 的支持，并且需要很大的内存来做各种复杂的图形学操作。但在嵌入式环境下，很多时候凑出足够的帧缓存都是很困难的，更不要说使用专用硬件进行图形学操作了。因此，用于嵌入式环境中的 GUI 需要有特别优化。

4.2　图形界面的使能和配置

RMP 通过提供一系列绘制功能来支持 GUI 界面，它的基本绘图功能包括了画点画线、几何图形绘制和位图抗锯齿渲染，而稍复杂的功能则包括了各类控件的绘制。这些绘制功能都是不记录 GUI 对象的状态的，也不采用面向对象设计，因此在使用时不消耗任何额外的 RAM 而仅仅消耗一定量的 ROM。

RMP 提供的绘图函数可以分为基本绘图函数、抗锯齿绘图函数和控件绘图函数。要使用它们，需要额外在配置头文件中打开相应的开关，还需要提供一些额外的底层宏。这些宏的列表如下：

表 4-1 使用内建嵌入式图形界面所需的宏定义

宏名称	作用
RMP_GUI_ENABLE	图形界面的总开关。将它定义为 1 以使能图形界面功能。 例子：#define RMP_GUI_ENABLE 1U
RMP_POINT(X, Y, COLOR)	画点函数的宏定义。一旦启用图形界面功能，则应该将这个宏定义为画点函数的函数名，而画点函数必须具备如下的原型： void foo(rmp_cnt_t X, rmp_cnt_t Y, rmp_ptr_t Color); 其作用是在屏幕上的某处画一个点。 例子：#define RMP_POINT RMP_Point
RMP_GUI_ANTIALIAS_ENABLE	抗锯齿绘制函数的总开关。将它定义为 1 以使能抗锯齿绘制功能。 例子：#define RMP_GUI_ANTIALIAS_ENABLE 1U
RMP_COLOR_25P(C1, C2) RMP_COLOR_50P(C1, C2) RMP_COLOR_75P(C1, C2)	颜色混合函数的宏定义。一旦启用抗锯齿绘制函数，则必须定义这些宏。三个宏都携带两个颜色作为参数，并返回一个颜色。 RMP_COLOR_25P 意味着将 C1 与 C2 做 25%+75%的混合， RMP_COLOR_50P 意味着将 C1 与 C2 做对半混合， RMP_COLOR_75P 意味着将 C1 与 C2 做 75%+25%的混合。
RMP_GUI_WIDGET_ENABLE	控件绘制函数的总开关。将它定义为 1 以使能控件绘制功能。 例子：#define RMP_GUI_WIDGET_ENABLE 1U
RMP_COLOR_WHITE RMP_COLOR_LGREY RMP_COLOR_GREY RMP_COLOR_DGREY RMP_COLOR_DARK RMP_COLOR_DDARK RMP_COLOR_BLACK	内建控件颜色的宏定义。一旦启用控件绘制函数，则必须定义这些宏。 RMP_COLOR_WHITE 为白色的编码定义，推荐#FFFFFF。 RMP_COLOR_LGREY 为淡灰色的编码定义，推荐#B8B8B8。 RMP_COLOR_GREY 为灰色的编码定义，推荐#E0E0E0。 RMP_COLOR_DGREY 为深灰色的编码定义，推荐#A0A0A0。 RMP_COLOR_DARK 为暗灰色的编码定义，推荐#787C78。 RMP_COLOR_DDARK 为极暗灰色的编码定义，推荐#686868。 RMP_COLOR_BLACK 为黑色的编码定义，推荐#000000。

RMP 提供的绘图函数在绘图时均不检查绘制范围的 X 和 Y 是否超出屏幕边界。这个责任是应用程序开发者负担的。

4.3 基本绘图函数

要使用基本绘图函数，要打开 RMP_GUI_ENABLE 开关，并定义 RMP_POINT 画点宏。

基本绘图函数的列表如下：

4.3.1 画直线

该函数在屏幕上绘制一条宽度为 1 的直线。绘制采用 Bresenham 算法进行光栅化。

表 4-2 画直线

函数原型	void RMP_Line(rmp_cnt_t Begin_X, rmp_cnt_t Begin_Y, rmp_cnt_t End_X, rmp_cnt_t End_Y, rmp_ptr_t Color)
返回值	无。
参数	rmp_cnt_t Begin_X 直线的起始点的 X 坐标。
	rmp_cnt_t Begin_Y 直线的起始点的 Y 坐标。
	rmp_cnt_t End_X 直线的终止点的 X 坐标。
	rmp_cnt_t End_Y 直线的终止点的 Y 坐标。
	rmp_ptr_t Color 直线的颜色。

4.3.2 画间断直线

该函数在屏幕上绘制一条宽度为 1 的间断直线。绘制采用 Bresenham 算法进行光栅化。

表 4-3 画间断直线

函数原型	void RMP_Dot_Line(rmp_cnt_t Begin_X, rmp_cnt_t Begin_Y, rmp_cnt_t End_X, rmp_cnt_t End_Y, rmp_ptr_t Dot, rmp_ptr_t Space)
返回值	无。
参数	rmp_cnt_t Begin_X 间断线的起始点的 X 坐标。
	rmp_cnt_t Begin_Y 间断线的起始点的 Y 坐标。
	rmp_cnt_t End_X 间断线的终止点的 X 坐标。
	rmp_cnt_t End_Y

	间断线的终止点的 Y 坐标。
	rmp_ptr_t Dot 间断线的点的颜色。
	rmp_ptr_t Space 间断线的空位的颜色。如果填入 RMP_TRANS，不绘制空位。

4.3.3 画矩形

该函数在屏幕上绘制一个矩形。矩形可以具备一个与填充颜色不同的轮廓颜色，可以填充也可以不填充。

表 4-4 画矩形

函数原型	void RMP_Rectangle(rmp_cnt_t Coord_X, rmp_cnt_t Coord_Y, rmp_cnt_t Length, rmp_cnt_t Width, rmp_ptr_t Border, rmp_ptr_t Fill)
返回值	无。
参数	rmp_cnt_t Coord_X 矩形左上角的 X 坐标。
	rmp_cnt_t Coord_Y 矩形左上角的 Y 坐标。
	rmp_cnt_t Length 矩形的长度。
	rmp_cnt_t Width 矩形的宽度。
	rmp_ptr_t Border 矩形的边界颜色。
	rmp_ptr_t Fill 矩形的填充颜色。如果填入 RMP_TRANS，不进行填充。

4.3.4 画圆角矩形

该函数在屏幕上绘制一个圆角矩形。圆角矩形的轮廓颜色与填充颜色一致，且本身必须被全部填充。

表 4-5 画圆角矩形

函数原型	void RMP_Round_Rect(rmp_cnt_t Coord_X, rmp_cnt_t Coord_Y, rmp_cnt_t Length, rmp_cnt_t Width, rmp_cnt_t Round, rmp_ptr_t Color)

返回值	无。
参数	rmp_cnt_t Coord_X 矩形左上角的 X 坐标。
	rmp_cnt_t Coord_Y 矩形左上角的 Y 坐标。
	rmp_cnt_t Length 矩形的长度。
	rmp_cnt_t Width 矩形的宽度。
	rmp_ptr_t Round 圆角矩形的圆角半径。
	rmp_ptr_t Color 圆角矩形的颜色。

4.3.5 画圆

该函数在屏幕上绘制一个圆。圆可以具备一个与填充颜色不同的轮廓颜色，可以填充也可以不填充。

表 4-6 画圆

函数原型	void RMP_Circle(rmp_cnt_t Center_X, rmp_cnt_t Center_Y, rmp_cnt_t Radius, rmp_ptr_t Border, rmp_ptr_t Fill)
返回值	无。
参数	rmp_cnt_t Center_X 圆心的 X 坐标。
	rmp_cnt_t Center_Y 圆心的 Y 坐标。
	rmp_cnt_t Radius 圆的半径。
	rmp_ptr_t Border 圆的边界颜色。
	rmp_ptr_t Fill 圆的填充颜色。如果填入 RMP_TRANS，不进行填充。

4.3.6 绘制单色位图

该函数绘制一个单色位图到屏幕。位图的长度必须为 8 的整数，位图必须按照行优先从左到右从高到低扫描的方式储存。在绘制时可以选择高位优先模式或者低位优先模式，当选择前者时，一个字节的最高位的点的 X 值在一个字节代表的各个像素中最小，选择后者则反之[1]。

表 4-7 绘制单色位图

函数原型	void RMP_Matrix(rmp_cnt_t Coord_X, rmp_cnt_t Coord_Y, const rmp_u8_t* Matrix, rmp_ptr_t Bit_Order, rmp_cnt_t Length, rmp_cnt_t Width, rmp_ptr_t Color)
返回值	无。
参数	rmp_cnt_t Coord_X 位图左上角的 X 坐标。
	rmp_cnt_t Coord_Y 位图左上角的 Y 坐标。
	const rmp_u8_t* Matrix 指向单色位图数据的指针。
	rmp_ptr_t Bit_Order 字节内的位序。RMP_MAT_BIG 为高位优先模式，RMP_MAT_SMALL 为低位优先模式。
	rmp_cnt_t Length 位图的长度，必须是 8 的倍数，否则绘制不会开始。
	rmp_cnt_t Width 位图的宽度。
	rmp_ptr_t Color 绘制位图所用的颜色。

4.4 抗锯齿绘图函数

要使用抗锯齿绘图函数，要打开 RMP_GUI_ANTIALIAS_ENABLE 开关，并定义 RMP_COLOR_25P、RMP_COLOR_50P 和 RMP_COLOR_75P 颜色混合宏。

目前该分类仅包括一个函数，它以 4 x FXAA 的抗锯齿模式绘制一个单色位图到屏幕。位图的长度必须为 8 的整数，位图必须按照行优先从左到右从高到低扫描的方式储存。在绘制时可以选择高位优先模式或者低位优先模式。在绘制时需要传入位图所在的图层的背景色以进行抗锯齿处理。要使用该函数还要求三个颜色混合宏已经被定义。

[1] 这和字节序的概念很相似，不过这里是一个字节内部的各个位的像素顺序

表 4-8 以抗锯齿方式绘制单色位图

函数原型	void RMP_Matrix_AA(rmp_cnt_t Coord_X, rmp_cnt_t Coord_Y, const rmp_u8_t* Matrix, rmp_ptr_t Bit_Order, rmp_cnt_t Length, rmp_cnt_t Width, rmp_ptr_t Color, rmp_ptr_t Back)
返回值	无。
参数	rmp_cnt_t Coord_X 位图左上角的 X 坐标。
	rmp_cnt_t Coord_Y 位图左上角的 Y 坐标。
	const rmp_u8_t* Matrix 指向单色位图数据的指针。
	rmp_ptr_t Bit_Order 字节内的位序。RMP_MAT_BIG 为高位优先模式，RMP_MAT_SMALL 为低位优先模式。
	rmp_cnt_t Length 位图的长度，必须是 8 的倍数，否则绘制不会开始。
	rmp_cnt_t Width 位图的宽度。
	rmp_ptr_t Color 绘制位图所用的颜色。
	rmp_ptr_t Back 位图所在图层的背景色。

4.5 控件绘图函数

要使用控件绘图函数，要打开 RMP_GUI_WIDGET_ENABLE 开关，并定义 RMP_COLOR_WHITE、RMP_COLOR_LGREY 、 RMP_COLOR_GREY 、 RMP_COLOR_DGREY 、 RMP_COLOR_DARK 、RMP_COLOR_DDARK 和 RMP_COLOR_BLACK 颜色宏。

4.5.1 绘制光标

该函数可以绘制各种 16x16 的光标到屏幕。

表 4-9 绘制光标

函数原型	void RMP_Cursor(rmp_cnt_t Coord_X, rmp_cnt_t Coord_Y, rmp_ptr_t Style)
返回值	无。

参数	rmp_cnt_t Coord_X 光标左上角的 X 坐标。	
	rmp_cnt_t Coord_Y 光标左上角的 Y 坐标。	
	rmp_ptr_t Style 绘制光标的样式。可选择如下项之一：	
	RMP_CUR_NORM	通常光标。
	RMP_CUR_BUSY	带有沙漏符号的忙光标。
	RMP_CUR_QUESTION	带有问号符号的光标。
	RMP_CUR_HAND	手形光标。
	RMP_CUR_TEXT	文字编辑光标。
	RMP_CUR_STOP	停止光标。
	RMP_CUR_MOVE	移动光标。
	RMP_CUR_LR	左右调整光标。
	RMP_CUR_UD	上下调整光标。
	RMP_CUR_ULBR	右下-左上调整光标。
	RMP_CUR_URBL	左下-右上调整光标。
	RMP_CUR_CROSS	十字准心光标。

4.5.2 绘制复选框

该函数在屏幕上绘制一个大小可变的复选框。

表 4-10 绘制复选框

函数原型	void RMP_Checkbox(rmp_cnt_t Coord_X, rmp_cnt_t Coord_Y, rmp_cnt_t Length, rmp_ptr_t Status)
返回值	无。
参数	rmp_cnt_t Coord_X 复选框左上角的 X 坐标。
	rmp_cnt_t Coord_Y 复选框左上角的 Y 坐标。
	rmp_cnt_t Length

	复选框的长度[1]。
	rmp_ptr_t Status 复选框的当前状态。填入 RMP_CBOX_CHECK，复选框被选中，0 则为不选中。

4.5.3 选中复选框

该函数选中一个已经绘制的复选框。

表 4-11 选中复选框

函数原型	void RMP_Checkbox_Set(rmp_cnt_t Coord_X, rmp_cnt_t Coord_Y, rmp_cnt_t Length)
返回值	无。
参数	rmp_cnt_t Coord_X 复选框左上角的 X 坐标。
	rmp_cnt_t Coord_Y 复选框左上角的 Y 坐标。
	rmp_cnt_t Length 复选框的长度[2]。

4.5.4 清除复选框

该函数清除一个已经绘制的复选框。

表 4-12 清除复选框

函数原型	void RMP_Checkbox_Clr(rmp_cnt_t Coord_X, rmp_cnt_t Coord_Y, rmp_cnt_t Length)
返回值	无。
参数	rmp_cnt_t Coord_X 复选框左上角的 X 坐标。
	rmp_cnt_t Coord_Y 复选框左上角的 Y 坐标。
	rmp_cnt_t Length 复选框的长度[3]。

[1] 同时也是宽度

[2] 同时也是宽度

[3] 同时也是宽度

4.5.5 绘制按钮

该函数绘制一个大小可变的按钮。

表 4-13 绘制按钮

函数原型	void RMP_Cmdbtn(rmp_cnt_t Coord_X, rmp_cnt_t Coord_Y, rmp_cnt_t Length, rmp_cnt_t Width, rmp_ptr_t Status)
返回值	无。
参数	rmp_cnt_t Coord_X 按钮左上角的 X 坐标。
	rmp_cnt_t Coord_Y 按钮左上角的 Y 坐标。
	rmp_cnt_t Length 按钮的长度。
	rmp_cnt_t Width 按钮的宽度。
	rmp_ptr_t Status 按钮的当前状态。填入 RMP_CBTN_DOWN，按钮被按下，0 则为弹起状态。

4.5.6 按下按钮

该函数按下一个已经绘制的按钮。

表 4-14 按下按钮

函数原型	void RMP_Cmdbtn_Down(rmp_cnt_t Coord_X, rmp_cnt_t Coord_Y, rmp_cnt_t Length, rmp_cnt_t Width)
返回值	无。
参数	rmp_cnt_t Coord_X 按钮左上角的 X 坐标。
	rmp_cnt_t Coord_Y 按钮左上角的 Y 坐标。
	rmp_cnt_t Length 按钮的长度。
	rmp_cnt_t Width 按钮的宽度。

4.5.7 弹起按钮

该函数弹起一个已经绘制的按钮。

表 4-15 弹起按钮

函数原型	void RMP_Cmdbtn_Up(rmp_cnt_t Coord_X, rmp_cnt_t Coord_Y, rmp_cnt_t Length, rmp_cnt_t Width)
返回值	无。
参数	rmp_cnt_t Coord_X 按钮左上角的 X 坐标。
	rmp_cnt_t Coord_Y 按钮左上角的 Y 坐标。
	rmp_cnt_t Length 按钮的长度。
	rmp_cnt_t Width 按钮的宽度。

4.5.8 绘制文字编辑框

该函数绘制一个大小可变的文字编辑框。

表 4-16 绘制文字编辑框

函数原型	void RMP_Lineedit(rmp_cnt_t Coord_X,rmp_cnt_t Coord_Y, rmp_cnt_t Length, rmp_cnt_t Width)
返回值	无。
参数	rmp_cnt_t Coord_X 文字编辑框左上角的 X 坐标。
	rmp_cnt_t Coord_Y 文字编辑框左上角的 Y 坐标。
	rmp_cnt_t Length 文字编辑框的长度。
	rmp_cnt_t Width 文字编辑框的宽度。

4.5.9 清除文字编辑框的一部分

该函数清除文字编辑框的一部分内容，可实现编辑框中的文字删除功能。

表 4-17 清除文字编辑框的一部分内容

函数原型	void RMP_Lineedit_Clr(rmp_cnt_t Clr_X, rmp_cnt_t Clr_Len, rmp_cnt_t Coord_Y, rmp_cnt_t Width)
返回值	无。
参数	rmp_cnt_t Clr_X 要清除的位置的起始 X 坐标。该坐标值为相对坐标值。
	rmp_cnt_t Clr_Len 要清除的长度。
	rmp_cnt_t Coord_Y 文字编辑框左上角的 Y 坐标。
	rmp_cnt_t Width 文字编辑框的宽度。

4.5.10 绘制单选按钮

该函数绘制一个大小可变的单选按钮。

表 4-18 绘制单选按钮

函数原型	void RMP_Radiobtn(rmp_cnt_t Coord_X, rmp_cnt_t Coord_Y, rmp_cnt_t Length, rmp_ptr_t Status)
返回值	无。
参数	rmp_cnt_t Coord_X 单选按钮左上角的 X 坐标。
	rmp_cnt_t Coord_Y 单选按钮左上角的 Y 坐标。
	rmp_cnt_t Length 单选按钮的长度[1]。
	rmp_ptr_t Status 单选按钮的当前状态。填入 RMP_RBTN_SEL，单选按钮被选中，0 为不选中。

4.5.11 选中单选按钮

[1] 同时也是宽度

该函数选中一个已绘制的单选按钮。

表 4-19 选中单选按钮

函数原型	void RMP_Radiobtn_Set(rmp_cnt_t Coord_X, rmp_cnt_t Coord_Y, rmp_cnt_t Length)
返回值	无。
参数	rmp_cnt_t Coord_X 单选按钮左上角的 X 坐标。
	rmp_cnt_t Coord_Y 单选按钮左上角的 Y 坐标。
	rmp_cnt_t Length 单选按钮的长度[1]。

4.5.12 清除单选按钮

该函数清除一个已绘制的单选按钮。

表 4-20 清除单选按钮

函数原型	void RMP_Radiobtn_Clr(rmp_cnt_t Coord_X, rmp_cnt_t Coord_Y, rmp_cnt_t Length)
返回值	无。
参数	rmp_cnt_t Coord_X 单选按钮左上角的 X 坐标。
	rmp_cnt_t Coord_Y 单选按钮左上角的 Y 坐标。
	rmp_cnt_t Length 单选按钮的长度[2]。

4.5.13 绘制进度条

该函数绘制一个进度条。

表 4-21 绘制进度条

函数原型	void RMP_Progbar(rmp_cnt_t Coord_X, rmp_cnt_t Coord_Y, rmp_cnt_t Length, rmp_cnt_t Width, rmp_cnt_t Style, rmp_cnt_t Prog, rmp_ptr_t Fore, rmp_ptr_t Back)

[1] 同时也是宽度

[2] 同时也是宽度

返回值	无。	
参数	rmp_cnt_t Coord_X 进度条左上角的 X 坐标。	
	rmp_cnt_t Coord_Y 进度条左上角的 Y 坐标。	
	rmp_cnt_t Length 进度条的长度。	
	rmp_cnt_t Width 进度条的宽度。	
	rmp_cnt_t Style 进度条的样式。可以是以下四种之一：	
	RMP_PBAR_L2R	从左向右增长的进度条。
	RMP_PBAR_D2U	从下向上增长的进度条。
	RMP_PBAR_R2L	从右向左增长的进度条。
	RMP_PBAR_U2D	从上向下增长的进度条。
	rmp_cnt_t Prog 进度条的进度，必须是一个 0-100 之间的数字。	
	rmp_ptr_t Fore 进度条的前景颜色，也即进度本身的颜色。	
	rmp_ptr_t Back 进度条的后景颜色，也即进度空白部分的颜色。	

4.5.14 改变进度条进度

该函数改变一个已绘制的进度条的进度。

表 4-22 改变进度条进度

函数原型	void RMP_Progbar_Set(rmp_cnt_t Coord_X, rmp_cnt_t Coord_Y, rmp_cnt_t Length, rmp_cnt_t Width, rmp_cnt_t Style, rmp_cnt_t Old_Prog, rmp_cnt_t New_Prog, rmp_ptr_t Fore, rmp_ptr_t Back)
返回值	无。
参数	rmp_cnt_t Coord_X 进度条左上角的 X 坐标。

rmp_cnt_t Coord_Y	
进度条左上角的 Y 坐标。	
rmp_cnt_t Length	
进度条的长度。	
rmp_cnt_t Width	
进度条的宽度。	
rmp_cnt_t Style	
进度条的样式。可以是以下四种之一：	
RMP_PBAR_L2R	从左向右增长的进度条。
RMP_PBAR_D2U	从下向上增长的进度条。
RMP_PBAR_R2L	从右向左增长的进度条。
RMP_PBAR_U2D	从上向下增长的进度条。
rmp_cnt_t Old_Prog	
进度条的原先进度，必须是一个 0-100 之间的数字。	
rmp_cnt_t New_Prog	
进度条的新进度，必须是一个 0-100 之间的数字。	
rmp_ptr_t Fore	
进度条的前景颜色，也即进度本身的颜色。	
rmp_ptr_t Back	
进度条的后景颜色，也即进度空白部分的颜色。	

4.6 图形界面设计指南

由于本库的功能相比全功能图形库功能极为受限，因此在设计时需要遵循一些基本原则，也需要一些设计技巧。这些设计技巧将在以下各节详细介绍。

4.6.1 扁平化设计

由于本图形库不支持窗口管理器，因此最好使用扁平化设计。扁平化设计也是目前的设计潮流。关于其具体概念和细节，可以参考相关的网站和文献资料，比如苹果公司的 UI 设计指导。

4.6.2 使用简单的背景和单色位图

由于本 GUI 实际上不支持图层叠加和半透明混合，因此其抗锯齿位图绘制功能要求传入固定的背景色来参与运算。如果采用非单色背景，那么该功能将无法使用。实际上在绝大多数界面中也用不到复杂的背景图片，因为往往一个深灰色背景就足够了。

同时，也要尽量避免使用单色位图以外的图形。这主要是由于灰度位图和彩色位图非常占用 Flash 空间。使用单色位图比起使用灰色或彩色位图可以节约大概 90%以上的存储空间。

4.6.3 字库的设计和实现

在使用本 GUI 库时，可以采用两种方法来实现字库。第一种方法是将使用的字做成单色位图，并且存储在微控制器的 ROM 中；另一种则是使用外挂的字库芯片。对于界面美观度要求较高，但是不要求显示全字符集的，建议使用前一种方法；对于那些要求使用全字符集的则要求使用后一种方法。

4.6.4 复杂控件的设计和实现

由于本 GUI 仅仅提供了那些最基本的控件，而这些最基本的控件在某些对美观程度要求较高的场合往往达不到要求，因此要自行设计控件。要显示自行设计的控件，可以使用本 GUI 提供的位图绘制功能对其进行绘制，在检测到控件状态变化之后去更新该控件到一个新的位图即可。

4.7 底层绘图函数的优化实现

与微处理器上常见的帧缓冲不同，微控制器的全部内存甚至不能满足一帧的缓冲需求，因此相搭配的屏幕往往自带显存。这种屏幕又称为 MCU 屏，每次改变显示内容都必须通过接口向屏幕传递数据，而这些接口甚至可能是 SPI、IIC 等性能低下的串行接口。在最糟的情况下，这些串行接口甚至是 GPIO 模拟出来的，使性能雪上加霜。因此，在完成同样绘图任务的前提下最小化接口操作次数是重中之重。如果每次都更新全屏，而且每画一个点都要重新设置 XY 坐标，会大大降低绘图性能。

针对上述问题，使用如下方法可以显著地提升微控制器的绘图速度：

4.7.1 区域控制法

在常见图形框架中，每次重绘都涉及整个屏幕或窗口，但实际上被改变的像素往往仅仅是这个屏幕或窗口的很小一部分。因此，只要更新发生变化的那一部分就可以，无须总是重绘所有元素。本方法可与其后所列的各种方法联用，以尽量消除无谓的屏幕更新。

4.7.2 宏内联法

RMP_POINT 会被非常频繁地调用，因此若能将其声明成一段简单的操作，就不要将其声明成函数。这可以节约函数调用的开销，并方便编译器执行深度优化。本方法可与其后所列的各种方法联用，以尽量消除函数调用开销。

4.7.3 共线自增法

RMP 图形接口的底层仅仅使用 RMP_POINT 一个宏，但这并不意味着每次绘制都必须按点来画。直线、矩形等常用绘图函数往往倾向于在一个方向上连续绘制，因此可以在绘制之前先设置屏幕的绘制坐标自增方向。这样，如果绘制的点都在同一行或者同一列，如果检测到本次 RMP_POINT 传入的坐标是

上次 RMP_POINT 坐标在自增方向上的延展，那么就不需要重新设置绘制坐标而只需要不断传输绘制数据即可。

4.7.4 写入合并法

部分单色显示屏每次都会写入连续的 N 个像素数据，无法在不读屏的前提下按照单个点来写屏，此时必须将多次 RMP_POINT 转化为一次写入：如果在连续的 RMP_POINT 调用中没有切换绘制的行，那么就等到同一条直线上的 N 像素都被写入后才写屏。如果最后一次传输凑不够 N 个像素，则要么（1）将显示屏上余下像素的颜色读回来凑够一组，要么（2）在了解背景颜色的前提下对余下的像素填充背景色。

4.7.5 块缓冲区法

虽然微控制器的内存无法放下完整的帧缓冲，但也许足以放下屏幕的一部分。一个 320x240 分辨率的 16 位真彩色屏幕需要 153kB 内存才能完整缓冲，但若只要缓冲其 1/4，则 38.4kB 的内存就足够了。因此，我们可以针对屏幕的不同分块分别绘制内容，并一次性上传该分块到屏幕。每绘制一个分块时，都需要对全屏进行模拟绘制，然而只有落在分块内的部分才会更新分块缓冲区。考虑到微控制器的计算速度远高于 I/O 接口操作速度，因此模拟数次全屏绘制可能是划算的。对于相同面积的屏幕，分块的数目越多，其内存占用就越少，但刷新全屏需要的模拟重绘次数也就越多，此时需要对分块的大小进行综合权衡。

部分屏幕还具备更新区域设置功能，在写一个完整的分块时可以完全省去坐标设置，从而进一步提升写屏速度。

4.7.6 调色板法

对于 16 位或 32 位真彩色屏幕，其每个像素都需要两个或四个字节来表达颜色。但并非每时每刻屏幕上都要显示每种颜色；很多时候同屏的颜色数量是很少的。此时，可以利用调色板法来压缩缓冲区空间。如果同屏发色数只有 16 色，则每一个像素用 4 位颜色代号表达就足够了，其中每一个颜色代号对应一种完整的 16 位或 32 位颜色。在向屏幕写入数据时，首先根据颜色代号查找真实的颜色数据，再将该颜色数据写入屏幕。该查找表还可以在程序运行过程中随时更改以使用不同的 16 色组合。

本方法可以与块缓冲区法联用以进一步减少内存占用。假设 TFT 彩屏的分辨率为 320x240，每次缓冲其 1/4，同屏发色数为 16 色，只需要 9.6kB 的内存就可以实现高效写屏，很多 16 位微控制器都具备这个量级的内存。假设 OLED 迷你彩屏的分辨率为 128x64，每次缓冲其 1/4，同屏发色数 16 色，只需要 1kB 内存就可以实现高效写屏，这几乎囊括了所有 8 位微控制器。

4.7.7 DMA 传送法

在接口支持 DMA 传送时，使用 DMA 能大大降低 CPU 的负担。在允许 DMA 传送时，推荐总是使用 DMA 传送，尤其是针对那些频繁产生中断的串行接口。

本方法可以与块缓冲区法巧妙地联用，此时需要准备两个块缓冲区：当 CPU 在一个块缓冲区进行全屏模拟绘图时，可以对另一个块缓冲区启动 DMA 传输，使 DMA 传输与缓冲区的更新同步进行。由于 CPU 的绘图一般比 DMA 传输要快[1]，因此缓冲区的大小可以设置为让 DMA 传输比 CPU 的全屏模拟绘图稍快一些完成。此时，在外界看来，微控制器对屏幕的更新是全速进行的，CPU 的全屏模拟绘图时间被完全隐藏，从而取得理论上的最高性能。

不过，本方法无法与调色板法联用，除非 DMA 硬件能够自动完成调色板的查找。

4.8 本章参考文献

无

[1] 串行接口的传输速率往往较低

第 5 章 形式化验证

5.1 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[1]，分别如下[1]：

表 5-1 EAL 等级及其描述

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

[1] EAL7+也列入在内

	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 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[1]，因此较为容易取得更高的认证水平。

5.2 系统的形式化规范

在 RMP 中，由于架构相关部分非常短小[2]，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，而 RMP 要支持的平台多达数十，从人力成本上讲不方便对其一一建立形式化模型进行验证。因此，主要的验证工作集中在 rmp_kernel.c 文件的 1500 行代码上。注释不计算在内，约有 1000 行之多。基于同样的多架构支持原因，对这些代码的正确性，我们仅验证至 C 语言语义的层面，并不进行二进制指令级别的验证。这意味着，最终可执行文件的正确性仍然需要依赖于编译器的正确性。由于在实践中编译器通常具备较高的认证等级，因此信任编译器的做法是合理的。当然，如果不希望信任编译器，也可以采用经形式化验证的 CompCert[2]编译器。

形式化验证并不能够完全杜绝软硬件系统错误操作的发生，尤其是硬件错误导致的问题[3]。对于硬件错误和硬件抽象层（Hardware Abstraction Layer，HAL）实现错误导致的系统失灵，RMP 的形式化验证是无能为力的。

5.3 形式化证明

将完成的后续工作。

5.4 其他文档

将完成的后续工作。

5.5 本章参考文献

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

[2] X. Leroy, "The CompCert verified compiler," Documentation and user manual. INRIA Paris-Rocquencourt, 2012.

[1] 多数 CC/CAPP/LSPP 等设计规范不适用

[2] 往往在 50 行代码以内

[3] 比如 CPU 或外设的硅片错误，这些错误的描述常常可以在各个厂商的芯片勘误手册内找到

第 6 章 移植 RMP 到新架构

6.1 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µC/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。

有关图形界面的底层接口，请参见第 4 章，在此不再赘述。

6.2 移植前的检查工作

6.2.1 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[1]，而不能在处理器内部通过硬件实现[2]。

6.2.2 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、IAR、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，也尽量避开了 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，要注意关闭低质量编译器的死代码消除功能和链接时优化功能，最好也要关闭低质量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 GCC -O2 或相当的优化水平。

6.2.3 汇编器

[1] 也即堆栈指针可以由用户修改，堆栈实现在内存中

[2] 这是 PIC 单片机等少数架构的典型实现方式

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。

6.2.4 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。

6.3 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 ARMv7-M 架构的文件夹名称为 Platform/A7M。其对应的头文件在 Include/Platform/A7M，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 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。

在移植到其他架构时，可以用 ARMv7-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。

6.3.1 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。

表 6-1 类型定义一览

类型	意义
rmp_s8_t	一个有符号八位整形。 例子：typedef char rmp_s8_t;
rmp_s16_t	一个有符号十六位整形。

	例子：typedef short rmp_s16_t;
rmp_s32_t	一个有符号三十二位整形。 例子：typedef int rmp_s32_t;
rmp_u8_t	一个无符号八位整形。 例子：typedef unsigned char rmp_u8_t;
rmp_u16_t	一个无符号十六位整形。 例子：typedef unsigned short rmp_u16_t;
rmp_u32_t	一个无符号三十二位整形。 例子：typedef unsigned int rmp_u32_t;

对于 RMP 而言，必须被定义的的类型定义一共有如下三个：

表 6-2 必要类型定义一览

类型	作用
rmp_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 unsigned long rmp_ptr_t;
rmp_cnt_t	计数变量的类型。这个类型应该被 typedef 为最长为一个处理器字长但至少为 32 位的有符号整数。 例子：typedef long rmp_cnt_t;
rmp_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 long rmp_ret_t;

6.3.2 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：

表 6-3 宏定义一览

宏名称	作用
RMP_EXTERN	编译器的 extern 关键字。当使用 C++编译器时，可以通过将其定义为 extern "C"来防止函数名重整。 例子： #define RMP_EXTERN extern
RMP_WORD_ORDER	处理器字长[1]对应的 2 的方次。比如，8 位[2]和 16 位处理器对应 4，32 位处理器对应 5，64 位处理器对应 6，依此类推。

[1] 按 Bit 计算

[2] 8 位处理器的指针至少是 16 位，因此在 RMP 看来它们是 16 位处理器

宏名称	作用
	例子： #define RMP_WORD_ORDER 5U
RMP_STACK_TYPE	栈的类型。对于递增满栈，请将其定义为 RMP_STACK_FULL_ASCEND；对于递减满栈，请将其定义为 RMP_STACK_FULL_DESCEND；对于递增空栈，请将其定义为 RMP_STACK_EMPTY_ASCEND；对于递减空栈，请将其定义为 RMP_STACK_EMPTY_DESCEND。 例子： #define RMP_STACK_TYPE RMP_STACK_FULL_DESCEND 取递减满栈定义。
RMP_STACK_ALIGN	栈的对齐要求。假设该值设为 X，则进入线程时的栈将保证被对齐到 2^X 字节。 例子： #define RMP_STACK_ALIGN 3U 将进入线程时栈的初始位置对齐到 8 字节。
RMP_STACK_ELEM	栈内部的元素类型；处理器每次常规数据压栈时会压入该类型的数据。 例子： #define RMP_STACK_ELEM rmp_ptr_t 栈内部的每个元素都是 rmp_ptr_t 类型。
RMP_STACK_STRUCT	初始化线程栈时使用的结构体名。 例子： #define RMP_STACK_STRUCT struct RMP_XXX_Stack 初始化线程栈使用的结构体是 struct RMP_XXX_Stack。
RMP_INIT_STACK_SIZE	初始线程堆栈大小，单位为字节。初始线程的栈是在 rmp_kernel.c 中内置声明的数组，因此如果某些架构对栈的内存地址有特殊要求[1]，则 rmp_kernel.c 的数据段必须被链接到满足这些要求的位置。 例子： #define RMP_INIT_STACK_SIZE 1024U
RMP_PREEMPT_PRIO_NUM	内核支持的抢占优先级的最大数量。这个数字必须大于等于 3。在实践中，如果内存不是特别紧张，推荐配置为处理器字长[2]的整数倍，但并非必须如此。通常而言，把这个值定义为处理器字长就可以了。 例子：

[1] 如必须位于地址空间的低 64k 等

[2] 按 Bit 计算

宏名称	作用
	#define RMP_PREEMPT_PRIO_NUM 32U
RMP_SLICE_MAX	内核允许的线程时间片或延时时间片的最大数量。这个数字不能超过 2^{N-1}-1（也即 RMP_MASK_INTMAX），其中 N 是 RMP_WORD_ORDER。 例子： #define RMP_SLICE_MAX 100000U
RMP_SEM_CNT_MAX	内核允许的信号量计数的最大数量。这个数字不能超过 2^{N-1}-1（也即 RMP_MASK_INTMAX），其中 N 是 RMP_WORD_ORDER。 例子： #define RMP_SEM_CNT_MAX 1000U
RMP_INT_MASK() RMP_INT_UNMASK()	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出错。如果使用该功能，RMP_INT_MASK()可以定义为掩蔽所有系统中断，而 RMP_INT_UNMASK()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_INT_MASK MASK(SYSPRIO) #define RMP_INT_UNMASK MASK(0x00U) 更详细的例子请参看 ARMv7-M 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。
RMP_MSB_GET(VAL)	求一个值的最高的二进制位位的位号，最低位的位号从 0 开始数起。比如，在 32 位处理器上的值 0x00001120 的最高二进制位是第 12 位。很多处理器能够提供诸如“数前导零（Count Leading Zeroes，CLZ）”的指令来帮助完成这一操作，此时可考虑用汇编编写这个宏的实现。部分编译器也提供前导零计算的内联函数。如果两者都不存在，则可将该宏定义为 RMP 提供的默认实现 RMP_MSB_Generic。 例子： #define RMP_MSB_GET(VAL) (32-__builtin_clz(VAL)) #define RMP_MSB_GET(VAL) RMP_MSB_Generic(VAL)
RMP_LSB_GET(VAL)	求一个值的最低的二进制位位的位号，最低位的位号从 0 开始数起。比

宏名称	作用
	如，在 32 位处理器上的值 0x00001120 的最低二进制位是第 5 位。很多处理器能够提供诸如“数后缀零（Count Trailing Zeroes，CTZ）”的指令来帮助完成这一操作，此时可考虑用汇编编写这个宏的实现。部分编译器也提供后缀零计算的内联函数。如果两者都不存在，则可将该宏定义为 RMP 提供的默认实现 RMP_LSB_Generic。 例子： #define RMP_LSB_GET(VAL) (__builtin_ctz(VAL)) #define RMP_LSB_GET(VAL) RMP_LSB_Generic(VAL)
RMP_STKSEG_ENABLE	控制堆栈段寄存器额外保存与恢复是否开启。如果开启，则 RMP 将额外定义 RMP_SS_Cur 变量以方便底层汇编保存和恢复堆栈段寄存器。在绝大多数架构上不需要开启此宏；有关堆栈段寄存器的相关信息，请参见 7.5.3.2。 例子： #define RMP_STKSEG_ENABLE 0U
RMP_ASSERT_ENABLE	控制断言宏是否开启。如果需要启用 RMP_ASSERT 宏以检测错误，请将其定义为 1，否则可将其定义为 0，此时断言将不被检查。 例子： #define RMP_ASSERT_ENABLE 1U
RMP_CHECK_ENABLE	控制参数检查是否开启。如果它被定义为 0，则系统调用在使用参数前不检查其合法性，且除特殊注明的情况外，系统调用将不再返回错误返回值。如果性能或代码体积的要求并不极端，请尽量开启参数检查。 例子： #define RMP_CHECK_ENABLE 1U
RMP_DBGLOG_ENABLE	控制内核打印宏是否开启。如果它被定义为 0，则内核打印宏无效化。 例子： #define RMP_DBGLOG_ENABLE 1U

6.3.3 汇编函数与宏

RMP 仅要求用汇编或内联汇编实现 5 个短小的底层汇编函数与宏。它们的名称和意义如下：

表 6-4 汇编函数与宏一览

函数名	意义
RMP_Int_Disable	禁止处理器中断。
RMP_Int_Enable	使能处理器中断。

_RMP_Start	启动初始线程。
RMP_YIELD	触发线程切换。
RMP_YIELD_ISR	在中断中触发线程切换。

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。

6.3.4 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：

表 6-5 系统中断向量一览

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。
线程上下文切换中断向量	处理线程上下文切换时使用。

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。

6.3.5 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：

表 6-6 其他底层函数一览

函数	意义
RMP_Putchar	打印一个字符到内核调试控制台。
_RMP_Lowlvl_Init	底层硬件初始化。
_RMP_Stack_Init	初始化某线程的线程栈。

6.4 汇编函数与宏的移植

汇编底层函数与宏的原型和移植要求详述如下。

6.4.1 RMP_Int_Disable 的实现

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。

表 6-7 RMP_Int_Disable 的实现

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

6.4.2 RMP_Int_Enable 的实现

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。

表 6-8 RMP_Int_Enable 的实现

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

6.4.3 _RMP_Start 的实现

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。

表 6-9 _RMP_Start 的实现

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

6.4.4 RMP_YIELD 的实现

该宏负责通知硬件抽象层进行上下文切换。该宏必须保证在它被调用之后、继续执行线程代码之前安排一次上下文切换。一般地，它需要软件触发可悬起的线程切换中断向量，这可以通过写入某个内存地址或者执行某条特殊指令实现。若中断没有被屏蔽，该软件中断在该宏退出前触发；若中断被关闭，

该软件中断必须悬起，且在开中断函数退出前触发。该宏必须含有一个编译器屏障[1]，以防止编译器将其后的代码优化到其前面。

表 6-10 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。

6.4.5 RMP_YIELD_ISR 的实现

该宏与 RMP_YIELD 类似，但它只会被带 ISR 后缀的中断发送函数调用。在该宏中，也需要触发软件触发可悬起的线程切换中断向量，其实现与 RMP_YIELD 完全相同。系统设计上将这两个宏分开是为了方便在移植时采取其它方式；有关这种方式，请参见 7.1.2。

表 6-11 RMP_YIELD_ISR 的实现

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

6.5 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上下文切换中断向量。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。

6.5.1 定时器中断向量

在定时器中断处理向量中，仅仅需要调用如下定时器中断处理函数：

表 6-12 定时器中断处理函数

函数原型	void _RMP_Tim_Handler(rmp_ptr_t Slice)
意义	执行定时器中断处理。
返回值	无。
参数	rmp_ptr_t Slice 在有节拍内核中，每次发生时钟中断时总是经过一个时间片，因此填充为 1；在无节拍内核

[1] 在那些支持链接期优化（Link-Time Optimization，LTO）的编译器中需要多加注意

	中，填充为上一次设置给定时器的时间片值。这个数字不受 RMP_SLICE_MAX 限制，但它不能超过 2^{N-1}-1（也即 RMP_MASK_INTMAX），其中 N 是 RMP_WORD_ORDER。

这个函数是系统实现好的，无需用户自行实现。有关无节拍内核和_RMP_Tim_Elapse，请参见 7.1.1。

6.5.2 上下文切换中断向量

上下文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. 将 CPU 的基本寄存器全部压入线程栈；
2. 将现在的线程堆栈指针存入变量 RMP_SP_Cur；
3. 切换到内核栈[1]；
4. 调用_RMP_Run_High 进行线程切换；
5. 将变量 RMP_SP_Cur 赋给现在的线程堆栈指针；
6. 将 CPU 的基本寄存器全部从线程栈恢复；
7. 切换到线程栈，退出中断。

表 6-13 线程上下文切换中断处理函数

函数原型	void _RMP_Run_High(void)
意义	执行线程上下文切换处理。该切换会更新变量 RMP_SP_Cur 和 RMP_Thd_Cur，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。从用户线程[2]发起的线程上下文切换也可以改用内嵌汇编的方法进行，此时需要在上下文切换过程中屏蔽中断。关于这种方法，请参见 7.1.2。

6.6 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。

6.6.1 底层硬件初始化

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

[1] 这一步是可选的，但推荐使用独立的内核栈和线程栈

[2] 而非中断向量

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。

表 6-14 _RMP_Lowlvl_Init 的实现

函数原型	void _RMP_Lowlvl_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。

6.6.2 线程栈初始化

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。

一般地，该函数将根据处理器的栈增长形式，在栈区头部[1]或尾部[2]选择一处地址，并在该处进行相应的初始化。如果该处理器还有栈对齐要求[3]，则该函数还要保证在弹栈结束后栈指针满足该对齐要求。各敏感寄存器[4]的初值必须符合该架构的二进制接口标准的规定，以防止在启动线程时出现错误。

为了方便编写该函数，RMP 提供了宏 RMP_STACK_CALC(PTR, CTX, STK, SZ)：它接受栈的起点 STK 和大小 SZ 作为参数，返回线程栈寄存器指针的真实初始地址 PTR 以及初始化线程栈时使用的上下文结构体指针 CTX；PTR 可以用作本函数的返回值，而填充 CTX 指向的结构体可完成栈区的初始化工作。

表 6-15 _RMP_Stack_Init 的实现

函数原型	rmp_ptr_t _RMP_Stack_Init(rmp_ptr_t Stack, rmp_ptr_t Size, rmp_ptr_t Entry, rmp_ptr_t Param)
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	rmp_ptr_t 线程栈的真实起始地址。线程第一次运行时，将从这里开始弹栈。
参数	rmp_ptr_t Stack 该线程的栈区的首地址。
	rmp_ptr_t Size 该线程的栈大小。

[1] 递增栈

[2] 递减栈

[3] 如 ARMv6-M、ARMv7-M 等

[4] 如各种段寄存器和运算状态寄存器等；可以通过进入调试模式并停留在 main 入口处来得到工具链对这些寄存器值的初值设置，也可以选择在_RMP_Start 中截获这些设置；如果涉及堆栈段寄存器，请参见 6.5.3.2

	rmp_ptr_t Entry 该线程的入口。
	rmp_ptr_t Param 给该线程传入的参数。

6.7 本章参考文献

无

第 7 章 附录

7.1 特殊内核功能的实现

在 RMP 中，有一部分功能没有被内核默认实现。但是在 RMP 中实现这些功能是可能的；具体的实现原理如下所列，如果用户需要这些功能则要自行实现。

7.1.1 无节拍内核的实现

无节拍内核要求系统具备一个大时长、高精度的定时器，并且由该定时器产生可变的调度器时间中断。时间片和定时器填充值之间的比例可以由实现者按需自由决定。为了方便实现无节拍内核，系统提供两个函数_RMP_Tim_Future 和_RMP_Tim_Elapse；如果不实现无节拍内核可以不必理会它们。

为了使定时器在最近的潜在超时时点产生中断，在每次发生定时器中断[1]、上下文切换[2]或新线程加入延时队列[3]后都需要检查当前线程的剩余时间，再检查最近超时的线程延时的剩余时间，然后选择两者中较小的一个设置成硬件定时器的下一次超时时间。

内核提供了函数_RMP_Tim_Future 用于得到该超时时间。有两种可能的特殊情况需要处理：（1）该超时时间为 0，此时说明有线程延时超时，需要立刻手动触发一个定时器中断，（2）如果这段时间的长度超过了定时器能支持的范围，只要将定时器的超时时间设置成允许的最大值即可；RMP 的设计保证只有实际经过的时间才会从线程剩余时间中抵扣掉，因此不用担心线程剩余时间丢失。

表 7-1 得到最近超时时间片数

函数原型	rmp_ptr_t _RMP_Tim_Future(void)
意义	得到距离未来最近一次超时的时间片数，可根据此设置系统定时器的下一次超时时间。
返回值	rmp_ptr_t 距离未来最近一次超时的时间片数。
参数	无。

在_RMP_Tim_Future 计算最近超时的线程延时的剩余时间前，需要保证时间戳 RMP_Timestamp 是最新的。否则，相应的计算就会出错。内核为此提供了函数_RMP_Tim_Elapse 用于更新该时间戳，

[1] 如定时器中断钩子 RMP_TIM_HOOK；定时器中断可能意味着（1）某个线程耗尽了自己的时间片，（2）某个线程延时的时间已到，（3）硬件定时器的最长可配置超时时间不够长而需要将一个定时分多段完成；在情况（1）中一定会发生上下文切换，在情况（2）中可能会发生上下文切换，在情况（3）中不可能发生上下文切换；在检测到上下文切换悬起时，不需要进行此次定时器编程，因为尾随的上下文切换将会完成这一工作

[2] 如上下文切换钩子 RMP_SCHED_HOOK

[3] 如延时插入钩子 RMP_DLY_HOOK

在定时器中断外[1]若要调用_RMP_Tim_Future 则必须先调用它。定时器中断中不需要调用它是因为定时器中断本身就会更新该时间戳。

注：观察_RMP_Tim_Elapse 调用时点可以发现，它要么出现在上下文切换向量中，要么出现在后续必然进行上下文切换的_RMP_Dly_Ins 中，而 RMP 中线程的时间片总是会在上下文切换后自动补满[2]。因此，在_RMP_Tim_Elapse 中无需更新当前线程的剩余时间片数量。这进一步导致在上下文切换中，_RMP_Tim_Elapse 只要求在_RMP_Tim_Future 之前调用，而不需要固定在切换之前调用[3]。

表 7-2 记录时间流逝数

函数原型	void _RMP_Tim_Elapse(rmp_ptr_t Slice)
意义	在计算定时器超时时间前记录当前线程已使用的时间片数。
返回值	无。
参数	rmp_ptr_t Slice 距离上次填充定时器以来经历的时间片数，该时间片数可通过读取定时器的计数寄存器，并计算其与设置初值之间的差值得到。这个数字不受 RMP_SLICE_MAX 限制，但它不能超过 2^{N-1}-1（也即 RMP_MASK_INTMAX），其中 N 是 RMP_WORD_ORDER。上述读取可以先停止定时器再行完成，因为接下来必然会在 RMP_SCHED_HOOK 中发生一次定时器重编程，定时器是否继续运行在此次读取后就无所谓了；RMP_DLY_HOOK 在调用本函数后还必须将定时器初值和计数值清零，以免后续 RMP_SCHED_HOOK 含有的本函数发生重复的时间抵扣。

为了进一步节约能源，当系统中只有初始空闲线程在运行，且当前没有正在延时的线程时，就连系统定时器的时钟源都可以关闭。系统特意提供了_RMP_Tim_Idle 来探知这一条件。要使用这一功能，需要保证先关闭中断，再使用该函数探测是否可关闭系统定时器时钟源，最后原子性地开启中断并同时进入合适的低功耗状态[4]。具体案例请参见 MSP430 的移植下的 CCS-MSP430G2553 工程。

表 7-3 查询系统定时器空闲状态

函数原型	rmp_ret_t _RMP_Tim_Idle(void)
意义	判断是否可以彻底系统关闭定时器及其时钟源以节约能源。
返回值	rmp_ret_t 若返回 0，可以关闭定时器及其时钟源，若返回-1 则不可以。

[1] 指 RMP_SCHED_HOOK 和 RMP_DLY_HOOK，不包括 RMP_TIM_HOOK

[2] 在 RME 中的时间片不会自动补满，而是需要用户态调度器手动传递时间片

[3] 在 RME 中，同名函数要求必须在向量开头调用，否则一旦切换完成，消耗的时间片就会被归咎给其它线程

[4] 如果 CPU 不提供原子性开启中断并关闭定时器时钟源的操作，或者中断来临时无法自动令时钟源起振，则此功能无法使用

参数	无。

值得注意的是，虽然无节拍内核的 CPU 功耗更低，但其实时性未必更好，因为它势必包含在上下文切换前的时间流逝登记，以及在上下文切换后对定时器的重编程。这都可能增加上下文切换临界区的长度，并对系统的最坏执行时间产生负面影响。在虚拟化环境中，这些缺点更是会成倍放大，因此在非低功耗环境下推荐选择有节拍内核。

7.1.2 线程上下文的保存和恢复

在通常情况下，本系统使用一个专用的线程上下文切换中断向量，并在且仅在其中进行线程的上下文切换。这使得上下文切换中断向量成为系统中唯一需要以汇编写成的中断向量，大大方便了系统的移植。这还降低了其它中断向量的中断延迟，因为在进入和退出其它中断向量时不需要压栈和弹栈全部的上下文。此外，在多个中断连续咬尾执行时，上下文的保存、恢复与切换仅在最后一个中断退出后才进行，这节约了很多开销。基于上述原因，推荐使用这种方法。

然而，这种方法并非没有缺点。它（1）需要一个额外的、软件可触发的、可悬起的中断向量[1]，（2）要求该向量必须在软件触发后几乎立刻进入[2]，（3）要求中断之间可以直接咬尾，也即上一中断退出后可以无缝接续[3]下一个，中间不得插入任何主程序指令，（4）增加了从中断向量激活某个线程的延迟，（5）增加了从线程发起上下文切换的延迟。在那些不满足（1）-（3）或不希望（4）与（5）发生的场合[4]，可以参考下面的实现思路。

首先，将 RMP_YIELD 使用汇编语言编写，并在其中完成如下操作：

1. 关中断；
2. 模拟处理器进入中断的行为，压栈中断会压栈的寄存器，且确保压栈的版本中中断开启；
3. 将 CPU 的基本寄存器全部压入线程栈[5]；
4. 将现在的线程堆栈指针存入变量 RMP_SP_Cur；
5. 切换到内核栈[6]；
6. 调用_RMP_Run_High 进行线程切换；
7. 将变量 RMP_SP_Cur 赋给现在的线程堆栈指针，切换到线程栈；
8. 将 CPU 的基本寄存器全部从线程栈弹出；
9. 调用中断返回指令，开中断并返回；
10. 调用普通返回指令，返回原程序流执行。

---

[1] 如 ARMv7-M 的 PendSV

[2] 触发后立刻进入中断，或触发到进入中断之间的指令数可控

[3] 部分 CPU 在中断返回后必须要执行一条主程序指令才能响应下一个中断

[4] 如将线程当成合作运行的纤程或协程使用时，或者软中断的中断延迟很高时

[5] 指令指针 PC（或 IP）需要一些特殊处置，压栈时不能压入它的原值，而要压入步骤 10 的返回点

[6] 这一步是可选的，但推荐使用独立的内核栈和线程栈

然后，用汇编语言编写中断向量的头尾，并在其入口都加上寄存器的保存（上记步骤 3、4、5），出口都加上寄存器的恢复（上记步骤 7、8、9），并在出口执行之前检测 RMP_Sched_Pend 是否置位。如果置位，执行线程切换（上记步骤 6）。RMP_YIELD_ISR 宏则无需定义。

使用本方法时，需要保证 RMP_YIELD 产生的上下文可以被中断向量的出口恢复，且中断向量的入口产生的上下文也可以被 RMP_YIELD 恢复，尤其是在那些进入中断时会进行硬件压栈的架构上。对于不提供专用上下文切换中断的架构，RMP 的官方移植均采取这种方法，并且会将上述汇编程序段封装成宏。所有的系统中断[1]均必须使用该宏来保存和恢复上下文，而透明中断[2]则无此要求。具体例子请参见 MSP430 的移植。

注 1：虽然我们可以在 RMP_YIELD 调用后都添加对 RMP_Sched_Pend 清零的循环等待，并待其清零后才继续执行代码以规避（2）中对几乎立刻进入的需求，但其他需求仍然无法简单规避。因此，在不满足（1）-（3）的架构的 RMP 的官方移植中不采取这个做法。

注 2：虽然关中断可以推迟到压栈完成后才执行以降低总关中断时间，但若在 RMP_Thd_Yield 引起的 RMP_YIELD 的压栈完成后[3]系统中断立刻来临，则线程的上下文会被保存两次。类似地，开中断可以提前到弹栈之前就执行，但这样做比推迟关中断更加可怕：打断 RMP_YIELD 弹栈过程的系统中断会再次压栈，且此次压栈在未来可能仍由 RMP_YIELD 负责弹栈；然而，RMP_YIELD 可能会在来得及弹栈之前再次被中断打断，从而造成理论上无限的栈用量[4]。为了避免上述问题，RMP 的官方移植总会先关中断再进行寄存器压栈，且在弹栈完成之前不会开启中断。

注 3：考虑到中断向量未必会在每次被触发时都请求上下文切换，因此在中断向量的入口与出口激进地保存和恢复一切寄存器是不必要的。有两个可能的优化：（1）在入口仅保存那些函数调用约定中调用者保存（Caller-Save）的寄存器，然后立刻执行中断向量，待执行完毕后在向量末尾根据 RMP_Sched_Pend 判断是否需要切换上下文。如果确需上下文切换，则再压栈剩余的寄存器并切换，否则则直接弹栈函数调用约定中调用者保存的寄存器并退出中断。（2）进一步地，在向量末尾判断退出后是否还有其它悬起的中断，若是则可将上述判断推迟到最后一个向量退出时才执行，本次无论 RMP_Sched_Pend 是何值都不执行上下文切换[5]。但是，这两个优化（1）需要适应各种编译器的潜在的不同的调用约定，（2）增加硬件抽象层的复杂程度，（3）仅提高平均效率而恶化最坏执行时间，因此在 RMP 的官方移植中不予实现。

注 4：在 RMP 的官方移植中，RMP_YIELD 与中断向量的出入口总是使用两套独立的实现，无论它们有多么相似也不复用任何代码。这是因为（1）我们不能假设所有汇编器都允许包含汇编头文件；在允许的场合，我们提供一个可供用户包含到中断向量汇编文件中的.inc 文件，该.inc 文件也可被架构的底层汇编文件包含，此时可以实现复用；在不允许的场合，提供一个可供用户手动粘贴到中断向量汇编文件中的.cpy 文件，此时无法实现复用；（2）这两套实现在某些架构上大相径庭，无法复用，比如某些

[1] 会调用_ISR 系列函数的、会访问操作系统临界区的中断

[2] 不会调用_ISR 系列函数的、与操作系统无关的中断

[3] 由 RMP_Sched_Unlock 引起的 RMP_YIELD 调用无此问题，因为此时调度器已锁、中断已关

[4] 在实践中几乎不可能，因此在不严肃的场合无需理会

[5] 以达到类似 ARMv7-M 的 PendSV 的效果

架构在进入和退出中断时会由硬件执行栈对齐等特殊操作[1]，而 RMP_YIELD 必须用另辟蹊径来模拟这套操作。总的来说，与其仅针对部分架构和工具链使用一套复用实现，不如对所有架构都使用两套独立实现；这能够保证所有底层心智模型的整齐划一。

7.1.3 协处理器上下文的保存和恢复

协处理器指浮点处理器、向量处理器等 CPU 扩展模块，它们也有自己的上下文；如果多个线程要并发共享同一个协处理器，则必须在线程上下文切换时一并保存和恢复该协处理器的上下文。为此，RMP 提供了 RMP_CTX_SAVE 和 RMP_CTX_LOAD 两个钩子。在保存钩子内，先要探测当前线程是否使用了协处理器，如果是则保存其上下文；在恢复钩子内也要先探测协处理器是否被使用，如果是的话则恢复其上下文。如果无法探测该协处理器是否被使用，则必须总是保存和恢复其上下文。

由于这两个钩子会在进入内核的 C 语言环境后被调用，在具备独立内核栈的移植中可以直接利用 RMP_SP_Cur 来将上下文保存在线程栈上；RMP 的所有官方移植都保证使用独立的内核栈。在内核共用线程栈的移植上则必须将协处理器上下文保存在额外的结构体中，否则可能会覆写内核返回地址而导致出错。

对于绝大多数架构，RMP 的官方移植中已经在栈上实现了其协寄存器的上下文保存和恢复。对于这一部分架构，不需要使用钩子函数就可以保存和恢复其浮点寄存器的上下文。针对不同的浮点处理器配置，可能需要在汇编文件中引用不同的保存和恢复例程。

注 1：在 RMP 的官方移植中，一切通用寄存器在内的上下文都被放在线程栈上。这是因为（1）几乎所有的处理器在进入中断时都至少会将 PC 和 FLAG 压栈[2]，部分处理器还会强制压栈一部分通用寄存器[3]，如果要将一切上下文都放进结构体中则需要拷贝这些寄存器到结构体，于性能不利，且（2）若将未自动压栈的上下文单独保存进结构体中，则线程启动前既需要需初始化线程栈中所含的 PC 和 FLAG，又需要初始化结构体，于软件工程不利。

注 2：在 RMP 的官方移植中，为了使所有浮点处理器配置情况都能收录收进一个汇编文件中，对浮点处理器的操作一律采用机器码。这可以在浮点处理器不存在时规避汇编器报错；RMP 会根据配置情况引用正确的保存和恢复例程，因此不会引起未定义指令异常。

注 3：在 RMP 上，推荐的协处理器使用方法是仅让一个线程使用它，这样就可以不必保存和恢复它的上下文，从而提高系统的效率和实时性。对于极少数特殊架构[4]，其协处理器的上下文由硬件全权控制，无法被软件保存和恢复，此时请务必保证系统中只有一个线程使用该协处理器。

7.1.4 线程内存保护的实现

在微控制器中，通常利用内存保护单元（Memory Protection Unit，MPU）来实现内存保护。与能进行地址翻译的内存管理单元（Memory Management Unit，MMU）不同，MPU 仅能限制物理内存的

[1] 如 ARMv7-M-RVM 的移植等

[2] 仅有的例外是 MIPS 和 RISC-V

[3] 如 Cortex-M 系列、C28x 等

[4] 如 DSPIC 系列的 DO 指令相关寄存器

访问权限，无法进行地址翻译。MPU 由一系列的寄存器组成，这些寄存器里面填充着各个被保护的物理内存区域的访问属性。每个线程可以拥有一套不同的 MPU 寄存器设置，并以此来限制自身对内存的访问。

推荐的实现是将 MPU 看作协处理器，也在 RMP_CTX_SAVE 和 RMP_CTX_LOAD 内进行其上下文的保存和恢复。需要注意的是，每个线程的 MPU 设置都应当允许读写 RMP 的内核内存和执行 RMP 的代码；这是因为 RMP 没有真正的用户态和内核态，而且所有的系统调用都被直接实现为函数调用。正因如此，基于 RMP 实现的内存保护仅能增强系统的功能安全性[1]，但不能提供任何信息安全性。

7.1.5 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[2]。

7.2 RMP 中已知的影响实时性的因素

在 RMP 中已知的影响实时性的因素只有两个，分列如下。这两个因素在实际的应用程序中都很好避免；通常而言，如果因为这些因素使系统的实时性受到损伤，应用程序就不是良好设计的。

7.2.1 延时队列

RMP 使用一个延时队列管理所有的延时。这个延时队列从前向后，各线程的超时时间依次增加。这样，每次时钟中断，RMP 都只需要检查队列中的第一个线程即可。但是，这个设计要求每当插入线程时都要遍历队列找到合适的插入位置以使得线程超时时间仍然递增。这个遍历的最坏时间复杂度对队列中已有的线程数量是 O(n)的[3]。从数学上可以证明，不存在一种数据结构能够使得定时器的所有操作总为 O(1)，因此这必然要求应用程序不要特别大量地使用延时等待或延时阻塞。

此外，RMP 提供的延时总是系统级的，低优先级线程的延时操作也需要锁调度器进行处理，这可能会干扰高优先级线程的运作。如果这会造成问题，更推荐用户使用有限的几个线程实现软定时器功能，并分别服务不同优先级线程的延时。

7.2.2 删除内核对象

线程邮箱和信号量等部分内核对象可以拥有一个等待队列，而删除该内核对象时需要将该等待队列中的线程从该队列中一一去除。如果某个内核对象上有过多线程在等待，则删除内核对象可能导致过久的锁调度器时间。

7.2.3 锁调度器和关中断

---

[1] 又叫可靠性；如探测栈溢出等常见事故

[2] 在 ARM 上，WFI 和 WFE 指令可以达到这个效果，而在 MSP430 上则必须修改 STATUS 寄存器的某些位

[3] 有 O(logn)的数据结构存在，但 RMP 的应用场景一般不需要如此重量级的实现

RMP 提供了锁调度器和关中断的接口。但是，这对系统实时性是有显而易见的损害的，因此在应用程序中应当尽量避免此类操作。比如，如果已知只有一个线程会在某内存池中做操作，那么再分配和释放内存时就可以不用锁调度器。

7.3 降低 RMP 的中断延迟

为了保证临界区不被入侵，RMP 在进行特定操作时需要锁调度器，而它的最简单的实现是关闭总中断。然而，这样做会使整个系统失去对外界事件刺激的即时响应，即使是那些与系统无关的透明中断[1]也会遭到禁用，导致从刺激源发生到中断向量执行之间的延迟变长[2]。在实时性要求较高的机电设备或接口模拟等应用场合，这种行为是不允许的。为此，RMP 提供了一系列方法来降低或彻底消除由临界区导致的中断延迟。无论使用哪种方法，系统中断[3]都必须使用 RMP 提供的上下文保存宏，而透明中断则只要求保存和恢复任何会被它们修改的寄存器，且系统中断的抢占优先级必须被设为相同以使其不能互相嵌套。

7.3.1 按优先级掩蔽系统中断

如果架构[4]提供了掩蔽某优先级以下中断的功能，且系统中断的优先级都在某值以下，就可以通过掩蔽优先级在该值以下的中断来达到仅禁止系统中断而不禁止透明中断的效果。在这些架构，RMP 的官方移植均使用掩蔽来替代禁止总中断，天然地不会引入任何中断延迟。配置头文件中的掩蔽等级则可供用户视具体情况调整。

7.3.2 按中断源掩蔽系统中断

在那些无法按优先级掩蔽中断的架构上，也可以通过依次禁止各系统中断源来达成和按优先级掩蔽相同的效果。不过，由于无法预测用户要使用哪些系统中断源，RMP 无法提供统一的方案：官方移植中的 RMP_INT_MASK()和 RMP_INT_UNMASK()宏都会操作总中断，RMP_YIELD 也会在进入和退出时操作总中断。

为了达到按中断源掩蔽中断的效果，需要对官方移植做如下修改：

1. 将 RMP_INT_MASK()和 RMP_INT_UNMASK()宏分别指定为对各系统中断源的屏蔽操作和解除屏蔽操作。

2. 在进入中断后，保存中断返回地址[5]，并立刻开启总中断。这只需要在调用 RMP 的官方上下文保存宏之前插入一条开启总中断的指令即可，这样关闭总中断的时间最多只有一条指令。如果硬件不提

---

[1] 不会调用_ISR 系列函数的、与操作系统无关的中断

[2] 这里的中断延迟指的是硬件中断延迟，它和 RTOS 的中断延迟是两回事；后者一般是指从中断中进行发送到对应的接收线程被激活的延迟，是 RTOS 的固有值，无法被降低

[3] 会调用_ISR 系列函数的、会访问操作系统临界区的中断，包括时钟中断和专用上下文切换中断

[4] 如 Cortex-M 系列等

[5] 绝大部分架构在进入中断时会自动将返回地址压栈，如 RISC-V 等则需要手动压栈后再开启中断，否则透明中断可以在返回地址还没来得及保存之前就覆写它而导致系统中断无法正确返回其原有上下文

供将所有系统中断都设置为相同的抢占优先级的功能，则需要对 RMP 提供的宏进行一定修改，以在开启总中断之前屏蔽各系统中断源，并在中断返回之前（1）禁用总中断，（2）解除对各系统中断源的屏蔽，以及（3）原子性地使能总中断并从中断中返回[1]。

3. 在 RMP_YIELD 的开头和结尾加上对系统中断源的屏蔽操作和解除屏蔽操作。

7.3.3 委托操作给软件中断

在系统中断源较多或者中断控制器配置不便的场合，对各系统中断源一一屏蔽和解除屏蔽会造成巨大的移植和执行负担。此时，如果处理器具备可悬起的软件中断源，或者软件可触发的多余的外设中断源[2]，则更高优先级的中断可以选择将部分操作委托（Delegate）到它内部执行，此时该软件中断是系统中断，而更高优先级的中断则是透明中断，大大降低了系统中断源的数量。如果中断发生得较频繁，这种做法还有降低处理器负担的优点，因为透明中断只需要保存和恢复它们使用的寄存器[3]。不过，对于被委托的操作而言，其本体的中断延迟时间和到线程的中断延迟都变长了，因此本方法有利有弊。

为达到委托操作的效果，需要对官方移植做如下修改：

1. 按照 7.3.2 中的方法，将包括该软件中断源在内的一切系统中断添加进按中断源掩蔽的对象中。
2. 在透明中断中，向某数据结构记录要进行的系统操作，并触发该软件中断源。
3. 在该软件中断中，从某数据结构读取系统操作，并代替透明中断完成它。

在该方法中，数据结构的设计是一个难点，因为该数据结构可能会被透明中断和软件中断同时修改。如果数据结构是较为简单的计数器等，可以使软件中断保留一个本地副本，并计算本地副本和计数器当前值的差值来决定应当进行多少次操作；计数器的原子性读取则可通过循环读取直至稳定的方法实现。如果数据结构需要记录复杂的额外信息，则可以采用原子操作指令访问它，甚至可以使用无锁（Lock-free）或无等待（Wait-free）数据结构；在处理器不提供这种操作时，也可以通过短暂地禁用对应的透明中断来达成相同的目的。有关互斥方案的设计请自行参阅相关资料，在此不做过多介绍。

7.3.4 委托操作给定时器中断

在找不到或不方便使用软件可悬起中断源又希望进行操作委托的场合，还可以选择将操作委托给定时器中断。它的原理和委托给独立的软件中断是类似的，只是（1）定时器中断负责处理委托，（2）定时器中断不由透明中断触发而由定时器硬件触发，且（3）系统必须是有节拍内核。

在实现定时器委托时，需要对官方移植做如下修改：

1. 按照 7.3.3 中的方法处理定时器中断这个唯一的系统中断源。
2. 在透明中断中，向某数据结构记录要进行的系统操作。
3. 在系统嘀嗒钩子 RMP_TIM_HOOK 中，从某数据结构读取系统操作，并代替透明中断完成它。

---

[1] 中断返回指令可以做到

[2] 外设的中断源一般总会富余，此时可以拿来做此用途

[3] 在那些具备多组寄存器的架构上，透明中断可以使用其它寄存器组以彻底绕过保存和恢复

此方法相当于将每个时间片内的所有中断操作都推卸给了定时器中断，因此这些操作可能被延迟很久；只有当线程并不需要在立即获知中断发生时才能使用这个方法。如果这些操作是通知性质而非计次性质，则可将多次操作合并为一次操作，此时能获得可观的性能提升[1]。

7.3.5 限制不关键中断对关键中断的干扰

在混合关键度系统中，部分透明中断源的关键性不高但优先级却很高[2]，如果不对其中断频率加以限制，必然会消耗大量的 CPU，甚至干扰关键但相对不紧急的其它中断的执行，造成它们的中断延迟的不受控的增长。

为了限制这些透明中断源的频率，需要采取以下措施：

1. 为每个透明中断源准备一个计数器，每发生一次中断则原子性地将其自减一，当计数器递减至零时自行关闭中断源。

2. 每当定时器中断或关键中断发生时，原子性地为该计数器（1）赋恰当的初值或者（2）在其未达上限之前增加一个定值，并打开对应的中断源。

此时，每个透明中断源都相当于被限制在了一个可推迟服务器（Deferrable Server）或偶发服务器（Sporadic Server）内，在一定的时间段内只能被触发有限的次数，如果有多于这个次数的触发，则中断会丢失而不予响应。这样一来，低优先级但高关键度的中断所受的干扰就是可控的。

7.4 缩减 RMP 的存储器占用

在内存特别宝贵的场合，有必要小心配置 RMP 以节省本就不多的 RAM 和 ROM。此种状况多见于 MSP430 等 16 位单片机和 Cortex-M0 等低端 32 位单片机上。在实践中，可以采用如下手法缩减 RMP 的内存占用：

7.4.1 降低抢占优先级的数量

不同于其他的操作系统，RMP 的线程邮箱是内建于线程之中的零成本功能，而其它功能在不使用时则不占用任何存储。因此，RMP 的编译时选项中只有一个会决定其内核的 RAM 占用，那就是 RMP 支持的优先级个数[3]。由于 RMP 的每个优先级都使用一个双向循环链表来管理，因此每个优先级都会消耗 2 个机器字长的 RAM。在一些微型系统上，配置过多的优先级是完全没有必要的，因此可以将 RMP 支持的优先级调整成 3 个来最大限度节省 RAM。在这种用例下，RMP 本身的 RAM 消耗甚至可以被挤压到 128 字节以内[4]。

7.4.2 调整厂商提供的库

---

[1] 尤其是针对缓冲区较深的网卡、串口等设备，可以等到接收到一定数据量时再行批次处理

[2] 紧急而不重要，偶尔丢失中断无所谓，如遥控解码、流媒体传输等

[3] 也即 RMP_PREEMPT_PRIO_NUM

[4] 如果不算内核中断响应用堆栈，那么实际大小是 64 字节

某些厂商提供的 HAL 库和启动文件可能会静态分配一些堆和栈[1]。在使用 RMP 的场合，厂商分配的堆是完全没有必要的，可以置 0；而厂商分配的栈则会被 OS 内核在响应中断时使用[2]，因此只要确认中断处理程序的最大栈消耗，然后将厂商分配的栈大小设置到比这一数字稍多[3]即可。此外，厂商库中用不到的文件可以不添加进工程，这也有助于减少 RAM 和 ROM 用量。如果厂商库占用的存储器不可忍受，那么也可以选择直接操作外设寄存器。

7.4.3 调整编译器选项

通过调整编译器选项，也有可能减小 ROM 和 RAM 占用。调整编译器选项为代码体积优化对减小 ROM 占用尤其有效，对减小 RAM 占用有时也有效果。

7.4.4 不使用动态内存管理

RMP 内建了一个动态内存分配器。该分配器在小内存设备上不推荐使用，因为分配器本身也会有一定的固定内存消耗。在小内存设备上，推荐静态分配所有的内核对象和变量。

7.5 移植时的推荐规则

在部分架构上，移植 RMP 可能会遇到两难抉择。下文将介绍这些抉择为何困难并给出推荐的处理规则。

7.5.1 处理器字长的推荐规则

在 32 位或更高的处理器上，指针的长度一般都是一个机器字长，这是因为这些处理器普遍能够以任何寄存器做间接寻址，而且机器字长的长度能够寻址整个地址空间，此时可以直接将 rmp_ptr_t 定义为一个机器字长。然而，对于字长更短的处理器，它们的（1）数据与代码空间可能是分开的，（2）指令集设计限制很大，（3）处理器字长不足以寻址整个地址空间，因此决定 rmp_ptr_t 的长度需要一些技巧。

7.5.1.1 8 位处理器的情况

在绝大多数 8 位处理器上无法或不需要移植 RMP，这是因为它们（1）内存资源太少，（2）为追求内存利用效率[4]，编译器采用局部变量静态覆盖技术，（3）为追求硅片面积效率，函数调用栈是由硬件实现的，（4）应用程序较为简单，使用 RMS 状态机就可以表达。在少数资源丰富且可以移植 RMP 的 8 位处理器上，几乎所有的编译器都默认指针长度为 16 位。这是因为 8 位仅能寻址 256 字节，这对任何应用程序来说都太少了。因此，rmp_ptr_t 至少要定义为 16 位。

---

[1] 比如 Cortex-M 系列微控制器的汇编启动文件的最前面

[2] 也即作为内核栈使用

[3] 一般推荐多几十字节

[4] 也有可能是由于架构缺乏 SP 相对寻址功能

极少数 8 位处理器甚至允许寻址 24 位内存空间，此时需要分成两种情况讨论。如果数据指针或函数指针中任意一个的内部二进制表示[1]高于 16 位，则在该架构上 rmp_ptr_t 需定义为 32 位，这虽然会浪费空间但别无他法。如果两个指针的内部表示都是 16 位，则在该架构上 rmp_ptr_t 可定义为 16 位。后一种情况是可能的，因为高于 16 位内存空间的寻址有两种实现方法：（1）由独立于指针的段寄存器实现，而若编译器的指针不包括段寄存器则指针就是 16 位[2]，且段寄存器的切换需要用户程序手动进行，（2）工具链保证将内核使用的的变量和代码链接到前 64KiB 地址空间[3]，后面的地址空间则可供绝大多数不被指针引用的变量和代码使用。有关段寄存器和内存模型的详细内容，请参见下两个小节。

7.5.1.2 16 位处理器的情况

绝大多数 16 位处理器都具备移植 RMP 的可能。这些处理器也可以分为只能寻址 16 位地址空间的类型，和能寻址更多地址空间的类型。对于前者，将 rmp_ptr_t 定义为 16 位即可；对于后者，则可视高内存空间的寻址方法，仿照 8 位处理器的方案处理。

7.5.2 段寄存器的保存和恢复规则

部分处理器具备段寄存器[4]，部分编译器也可能在某些条件下使用一个全局变量指针寄存器[5]。根据编译器对其使用风格的不同，段寄存器的上下文保存和恢复也可做不同处理。

7.5.2.1 段寄存器仅被启动代码设置

这是最容易应对的情况。此时，段寄存器仅在启动时被启动代码设置一次，之后便不再进行更改；这可能是由于程序较小，在一个段内即可放下。如果能确信程序的行为确实如此，则可以完全不进行段寄存器的保存和恢复。

7.5.2.2 段寄存器总在访存前临时更改

这是较容易应对的情况。此时，编译器不会对段寄存器的值做任何假设，而是在访存前临时将段寄存器指向需要的段。由于段寄存器的值可能发生变动，且中断可能插在段寄存器修改和段寄存器使用之间，因此需要对段寄存器做上下文保存和恢复。

7.5.2.3 段寄存器在函数调用前要求保持初值

这是最难应对的情况。此时，编译器不仅会在函数内部生成修改段寄存器的操作，还假设在进入包括中断向量在内的每一个函数之前段寄存器都是初始值；这一般是为了优化程序的效率，若启动代码能

---

[1] RL78 等部分架构的指针字面值是 20 位，但内部二进制表示却是 16 位

[2] 如 DSPIC 等架构

[3] 如 AVR 等架构

[4] 如 DSPIC 等架构

[5] 如 MIPS、RISC-V 等架构

将段寄存器在初始化时就指向访问最频繁的段，则可在函数调用之间省去段寄存器设置工作，并减小代码体积。那些访问其它段的函数则需要在它们的头尾保存和恢复段寄存器。

在此种情况下，不仅需要对段寄存器做上下文保存和恢复，还需要在_RMP_Start 中保存段寄存器的初值；在中断向量[1]中如果需要调用 C 函数，则必须在调用前将各段寄存器设置为_RMP_Start 保存下来的值。这种做法的优点在于，不论工具链和启动代码将这些值设置成什么，它们都必然被_RMP_Start 截获，这样就不需要从五花八门的链接器脚本中取得这些值。当然，如果工具链是 GCC 等开源标准品，也可以选择从链接器脚本中直接取得这些值。

部分工具链的段寄存器使用风格随着选择的内存模型而变化。因此，在有段寄存器的架构上，RMP 的几乎所有官方移植都采取上述方法以消除段错误风险，即便这可能是多此一举。只有一种例外：该架构只计划提供一种工具链支持，且该工具链明示了各段寄存器的初值为某固定值[2]。

7.5.3 内存模型的推荐规则

在部分 8 位和 16 位处理器上，由于有段寄存器的存在，程序可以选择使用一个或多个代码段（CODE）、数据段（DATA）或堆栈段（STACK），从而产生不同的内存模型（Memory Model）。当程序使用一个段时，其指针为 16 位，但其寻址范围被限制在一个段以内；当程序使用多个段时，其指针为 24 或 32 位，其寻址范围包括整个内存地址空间。部分架构甚至将同一种存储器按地址空间分为多种类别，每种类别的指针长度都各不相同。此外，代码段、数据段和堆栈段的默认指针类型是可以独立设置的：假设代码段的指针长度有 C 种，数据段的指针长度有 D 种，堆栈段的指针长度有 S 种，则内存模型的种类有 C×D×S 种。部分架构的段寄存器方案甚至到了五花八门的地步[3]，因为它们并未提供架构级的段寄存器，而是要求用户在外部使用分立逻辑配合 I/O 口实现存储器映射电路。

为了简化起见，RMP 不可能针对每一种情况都给出相应的移植，且 RMP 在设计时假设一切指针均为等长，这要求代码段、数据段和堆栈段的默认内存模型相同。这可能在移植时导致两难选择：如果总是不加考虑地选择最大的内存模型，则 rmp_ptr_t 是 32 位，大大增加了处理器的负担，尤其是程序规模较小时得不偿失；如果总是不加考虑地选择最小的内存模型，则 rmp_ptr_t 是 16 位，处理器负担较轻，但无法访问更多地址空间。为此，我们按照以下原则将处理器分为三类，并针对不同的类别分别给出推荐方案。

7.5.3.1 常用内存模型为大模型

小部分处理器[4]从诞生之初就常使用大内存模型，它们的绝大多数工具链默认 sizeof(void*)为 4。这是由于它们的存储器类别太多、部分类别的寻址范围又太少，就连工具链作者也难以应付大量组合带来的复杂性，只能采取长指针来逃避它。

---

[1] 和 RMP_YIELD 中，如果不使用专用线程上下文切换中断向量的话

[2] 如仅提供 GCC 工具链移植的 AVR 架构

[3] 如基于 6502 的红白机（Famicom/NES）的存储器管理器（又称 Mapper）

[4] 如 8051、80251 等架构

在这种处理器上，RMP 的官方移植采取大模型：它将 rmp_ptr_t 定义为 32 位，其中高 16 位就是段寄存器的值。段寄存器和相应的裸指针一并保存和恢复。

7.5.3.2 常用内存模型为小模型

大部分处理器仅在用户明示时才使用大内存模型，它们的绝大多数工具链默认 sizeof(void*)为 2，这是为了尽量提升程序性能敏感部分的效率。当程序需要访问更多内存时，程序员需要在变量和函数前使用 far 等关键字声明其为远符号，编译器会自动生成相应的段和跨段访问代码，使其在小模型下也能访问更宽广的内存地址。

在这种处理器上，RMP 的官方移植采取带段寄存器保存和恢复的小模型：它将 rmp_ptr_t 定义为 16 位，但会在上下文切换时切换段寄存器以支持线程对其它段的访问。用户必须确保所有的内核数据结构[1]都与内核的数据段是同一个，且所有的线程入口[2]都与内核的代码段是同一个；不过，一旦线程启动，它可以访问其它段的变量以及跳转到其它段去运行，不受这一限制。

在移植时，堆栈段需要视情况做特殊处理：（1）如果该处理器未提供堆栈段寄存器，或其组成的常见系统中 RAM 与 64KiB 相去不远[3]，则堆栈段无需特殊处理；（2）如果该处理器提供了段寄存器且其 RAM 总量常远超 64KiB[4]，此时可以将 RMP_STKSEG_ENABLE 宏定义为 1 以将堆栈段寄存器保存在线程控制块中。一旦该宏被启用，“创建线程”系统调用和“线程栈初始化”底层调用的格式均会发生改变，在 Stack 参数前会出现一个额外的 rmp_ptr_t Segment 参数以方便用户传入堆栈段寄存器的值。

注：在（2）中，堆栈段寄存器不能保存在栈上，因为从栈中恢复堆栈段寄存器需要弹栈，而弹栈又会用到堆栈段寄存器，从而造成悖论问题。

7.5.3.3 常用内存模型无法推测

一部分处理器的内存模型无法推测[5]，它们的绝大多数工具链虽默认 sizeof(void*)为 2，但允许用户编写自定义的段切换代码以配合片外定制电路实现跨段访问。

对于此类架构，RMP 的官方移植一律采取不理会段寄存器的小模型。用户必须对底层移植进行少许定制修改以适应其硬件结构。

7.5.4 移植兼容性的推荐规则

几乎所有微控制器厂商都会在一个基础指令集上搭配一系列可选的协处理器[6]以适应不同的细分市场。部分厂商甚至会增加老架构中寄存器的长度[7]来适应新的需求。在移植 RMP 时，一个简单的办法是

---

[1] 包括各对象控制块以及消息队列中的消息

[2] 不将它们声明为 far 即可

[3] 常见于微控制器：它们要么没有堆栈段寄存器，要么内存小于 64KiB 而使堆栈段寄存器总为固定值

[4] 常见于 8086 等通用微机系统

[5] 如 6502、Z80 等架构，但不包括 Rabbit 2000 等官方定义了段寄存器的架构

[6] 或者扩展指令集等

[7] 如 MSP430X 等架构

针对不同的协处理器组合编写独立的移植，但这样会导致各移植之间包含巨量的重复代码。因此，我们制定一些规则，并依此判断哪些协处理器组合应该被合并为一个移植。

7.5.4.1 链接能跑原则

这是决定合并移植的最重要原则。如果多种协处理器组合[1]的二进制中存在一个子集，该子集可以被链接到任何一种组合，且得到的最终二进制能够在该组合上运行，则这些组合可以合并为一个移植。此时，在各芯片的配置头文件中额外增加协处理器选择宏 RMP_XXX_COP_YYY，其中 XXX 是移植名称，YYY 是该扩展指令集，选中该宏则选中该协处理器组合。

协处理器之间的相容性可以分为“含”性、“与”性、“或”性、“非”性和“异或”性：对于 A 和 B 两个协处理器，“含”性意味着若 A 存在则 B 必须存在，反之未必成立；“与”性意味着若 A 存在则 B 必须存在，反之亦然；“或”性意味着 A 和 B 的存在性相互独立，互不干涉；“非”性意味着 A 和 B 中仅能存在一个，但可以都不存在；“异或”性意味着 A 和 B 中必须且仅能存在一个。不同组合之间的相容性由移植底层代码在编译时确认，并在用户选中错误的组合时通过#error 报告。通常而言，微控制器的协处理器都很简单，不涉及非常复杂的兼容性判断。

注：“链接能跑”原则除涉及各协处理器组合的指令公共子集关系之外，还涉及到主流编译器对各协处理器组合的处理方式。在以下情况中，即便存在公共的指令子集，软件上也可能互不兼容：（1）该公共子集图灵不完备以至于无法单独形成应用程序，（2）该公共子集无法在编译器中单独选择，也即无法通过编译生成仅含有该公共子集的二进制[2]，（3）不同协处理器组合的调用约定完全不同以至于无法以统一方式传递两个以内的参数，(4) 不同协处理器组合的内存模型完全不同以至于无法兼容，以及 (5) 不同协处理器组合要求的栈对齐字节数不同。如果出现情况（1）-（4），则认为不满足本原则[3]，必须为每种组合提供独立的移植；如果仅出现情况（5）则认为满足本原则，且提供的合并移植中的栈对齐字节数将是所有组合的可能最大值[4]。

7.5.4.2 心智统一原则

这是决定合并移植的次要原则。如果以下两条中的任意一条原则成立，则认为该协处理器组合具有统一的心智模型，则这些组合可以合并为一个移植：（1）厂商在其官方资料中明示这些协处理器组合是同一种处理器的衍生品[5]，（2）厂商虽未明示，但在技术手册中经常将这些协处理器组合以“合订本”的形式相提并论[6]。

---

[1] 含无协处理器的情况

[2] 汇编也许可以，但它的编写效率太低，因而毫无意义

[3] 比如 C54x 和 C55x 由于出现（3）、（4）的情况，被认为是两个独立的架构

[4] 比如 C64x、C67x、C64x+、C67x+、C66x 由于仅出现（5）的情况，被合并为 C6x 架构

[5] 如 ARMv7-M 的 FPV4_SP、FPV5_SP、FPV5_DP

[6] 如 DSPIC 的 PIC24、PIC33 及其他五花八门的衍生品

对于满足“链接能跑”原则，但不满足“心智统一”原则的协处理器组合[1]，RMP 的官方移植总是将其分开以维持其心智模型的独立性。

7.5.4.3 高度重复原则

这是决定合并移植的最次要原则。如果以上两条均成立，且协处理器组合之间的底层代码有相当程度的重复度，则这些组合可以合并为一个移植：高重复度说明（1）协处理器组合之间的相似度高，（2）这些协处理器组合对应的产品处于相同或相近的市场生态位。

对于满足“链接能跑”和“心智统一”原则，但不满足“高度重复”原则的协处理器组合，RMP 的官方移植总会将 C 代码合并在一个文件内，但汇编文件则可能分开提供。此时，在各芯片的配置头文件中也额外增加协处理器选择宏 RMP_XXX_COP_YYY，其中 XXX 是移植名称，YYY 是该扩展指令集，选中该宏则选中该协处理器组合。

7.6 厂商库的兼容性

在开发者时间极为宝贵的今天，绝大多数微控制器厂商都提供了固件库。通过这些易学易用的库函数访问片上外设要比使用寄存器读写容易得多。不幸的是，部分固件库的底层功能可能会与操作系统冲突，因而需要修改[2]。常见的冲突发生在中断以及延时两方面，其解决方案如下所述。

7.6.1 开关中断的兼容性

操作部分外设时，其读写序列必须在规定的时间之内一次性做完，在此期间被中断打断是不允许的。此时，固件库往往会先关中断以便完成操作，然后再开启中断。对于这些库函数，不要在系统锁调度器期间调用它们，否则它们会在调度器解锁之前不通知操作系统而开启中断，造成系统崩溃。如果无法确定这些库函数是否会在锁调度器期间被调用，可考虑将其开关中断函数更换为 RMP 提供的调度器锁定/解锁函数，并将 RMP_INT_MASK 和 RMP_INT_UNMASK 指定为掩蔽所有中断。

7.6.2 延时函数的兼容性

操作部分外设时，其读写序列往往需要等待外设就绪，因此步骤间常插入延时函数。固件库实现延时函数的方式是五花八门的，可以采用忙循环法、定时器查询法或定时器中断法，部分固件库中甚至针对不同的用途同时采取多种实现方式。

忙循环法的延时函数通过运行一段无用的指令序列来达成延时，它们的行为与一般代码无异，不需要做任何修改就能与 RMP 兼容。这种延时函数还能用于锁调度器或关中断的场合，因为它不依赖于中断系统。当然，如果能够确定固件库从不在锁调度器或关中断期间调用这种延时函数，可以考虑将它们重写，并使用更节约处理器性能的 RMP_Thd_Delay 替换之。

---

[1] 如 ARMv6-M 和 ARMv7-M

[2] 部分厂商考虑到了这一点，将默认的实现加上了“弱符号”属性，此时只要重写它们即可

定时器查询法的延时函数通过初始化定时器并反复查询定时器的情况来确定延时是否完成。它比忙循环法更精确，也同样能在锁调度器期间使用，但它们可能干涉 RMP 的节拍定时器。此时，该延时函数可能会修改节拍定时器的（1）计数器值，（2）计数上限值或者（3）工作频率，这都可能导致系统工作不正常。更有甚者还会在初始化时关闭定时器的中断，导致 RMP 的调度器失控。当上述情况发生时，可以重写该函数，并（1）使其利用其它的定时器，（2）通过忙循环法完成延时，或者（3）通过调用 RMP_Thd_Delay 完成延时。方法（3）是最节约处理器性能的方法，但并不总是可行，因为固件库可能在锁调度器或关中断期间调用延时；如果此时又没有其它定时器可用，则只能选择 RMP_Thd_Loop 提供的忙循环法完成延时。

定时器中断法的延时函数通过初始化定时器中断，然后在中断内递增某全局变量，且在延时时不断查询该全局变量来确定延时是否完成。这种延时函数的延时精度高、延时范围宽，而且还能够提供一个时间戳变量以供查询，但也可能干涉 RMP 的节拍定时器。不过，这种延时函数必然只能在中断开启时使用，因此可以用 RMP_Thd_Delay 进行简单替换。如果该延时函数产生的全局变量也被读取，则可以使用 RMP_Timestamp 替代它。

7.6.3 代码生成器的兼容性

为了使微控制器更加易用，部分厂商在库函数之上还额外提供了代码生成器。这些代码生成器的产出有可能与 RMP 起冲突。常见的冲突有如下三种：

1. 生成含有 main 函数的代码。为了兼容各厂商提供的启动代码对 main 函数的调用，RMP 在内核中声明了 main 函数。当代码生成器生成的框架与 RMP 一起编译时，main 函数就会被定义两次，并在链接时引发错误。鉴于这些 main 函数包含的大多是简单的硬件初始化代码，可以将该 main 函数直接改名或宏定义为 hw_main，然后将配置头文件中的 RMP_LOWLVL_INIT 定义成它。为了简化系统启动的心智模型[1]，RMP 并不像 RME 那样提供独立的 RME_Kmain，因此其它代码需要避让 main 函数。

2. 生成初始化看门狗、开启某些无关外设中断等会意外干扰系统运行的代码。最好的解决方案是配置生成器使其不生成该种代码；如果这不可能，则必须手动修改生成后的代码，除去这些干扰项，或者在系统启动和运行过程中加入额外操作[2]使它们不造成干扰。

7.7 RMP 对 C 语言标准和 MISRA 标准的背离

7.7.1 C 未定义行为：类型双关指针别名可以存在并不遵循严格别名机制

类型双关（Type-Punning）指针别名（Pointer-Aliasing）是一种广泛存在于 C 语言[3]实际应用中的用法，它允许两个不同类型的指针引用同一块内存。这在网络协议栈中是非常常见的，因为我们往往

[1] 在 main 函数进入后，部分存储器控制器、段寄存器等可能未初始化，且此时的执行并不处于线程环境中，进行某些内核操作可能会出错而且原因很难排查，而在一切都初始化好后再行操作则保证不会出现此类问题；此外，我们希望保证系统的入口总是 main 函数，这样就能直接用它的栈来初始化中断响应栈，利于适配各裸机工具链；鉴于上述两点，我们不采用其它 RTOS 普遍使用的独立内核入口函数设计

[2] 如喂看门狗等

[3] 指 C90，下同

需要组装或者拆解网络封包，变换其字节序，等等。在操作系统中这也很常见，尤其是将无符号整数指针与结构体指针来回转换以方便对对象的操作。不幸的是，C 语言的严格别名机制（Strict Aliasing Rule）并不允许两个不同类型的指针引用同一块内存，除非是兼容类型或者从其他类型转换到 char[1]，而且也没有通用的办法绕开这一点。通常而言有三种方法可以避免类型双关，它们分别是：

1. 使用 union 来进行不同类别之间的转换。不幸的是这在 C 中是被定义行为而在 C++中不是，而 RMP 的代码由于可能和使用 C++写成的某些库直接链接[1]而需要作为 C++被编译，因此仍然有可能是未定义行为。此外，此种方法对长度在编译时不确定的数组对象是束手无策的，除非牺牲潜在的性能来对它们进行逐一转换。最后，这样使用 union 不符合 union 的设计精神，属于对语言规则的滥用。

2. 使用 char*来进行操作，因为 char*总是可以和任何指针别名。不幸的是现存编译器对这一点的优化非常糟糕；某些相当优秀的编译器甚至也会把这些操作编译成逐字节访问的指令，对于低质量嵌入式编译器则更是如此。因此，这是一种不好的解决方案。

3. 使用 memcpy 函数在两个不兼容的指针间拷贝内容，操作完后再拷贝回去，寄希望于编译器能够探测这种行为并且优化掉重复的 memcpy。这巧妙地利用了 memcpy 的参数为 char*的特点。它带来的直接问题是，某些低质量嵌入式编译器可能不够智能，有时不会把 memcpy 函数优化掉，而导致生成的代码中实际调用了两次 memcpy，从而导致巨大的性能损失。而且一旦 memcpy 没有被优化掉，这可能会导致在函数内部声明的局部拷贝缓冲区被分配在栈上，大大增加栈消耗而导致堆栈溢出。最后，此类对 memcpy 的使用是不符合 memcpy 的设计意图的，属于对语言漏洞的滥用；C 标准对此类做法不持鼓励态度。

有鉴于以上三种方法的问题，RMP 并不使用这些方法，而是直接使用类型双关指针别名，而在编译时关掉相关的优化[2]。

7.7.2 C 未定义行为：一个字节的长度总是八位，一个 char 类型的长度总是一个字节

C 语言并未定义 char 的长度，它仅仅表示 char 必须等于系统中基本字符集的单个字符的宽度[1]，理论上讲一个 char 类型可以是 16 或 32 个二进制位[3]，或者甚至更多。现时的所有主流编译器和架构上 char 的长度都总是 8 个二进制位，因此我们在代码中也作此假设以避免不必要的“char 类型长度”配置宏。

事实上，RMP 不会使用 char 型的高于 8 位的二进制位，也不会按字节操作或以字节为基础单位衡量更长的数据类型[4]，因此在 char 型长度超过 8 位的前提下仍能正常运作[5]。不过，使用这类移植时必须小心：（1）rmp_u8_t 不再是字面意义的 8 位，而是可以表达 8 位的最小整数类型[6]，（2）本手册中所有的地址都是按字寻址，而且（3）本手册中所有的数据结构大小和内存长度都是指字数而非字节数。

---

[1] 如果不希望使用 extern "C"的话

[2] 如 GCC 的-fno-strict-aliasing 选项

[3] 某些早期数字信号处理器，如 C28x 的编译器会这样做；但现代数字信号处理器很少再这样做

[4] 使用的是按位操作和按 char 衡量，这在所有处理器上都是通行的

[5] 见 C28x 的移植

[6] 类似于 stdint.h 头文件中的 uint_least8_t

7.7.3 C 未定义行为：指针的实际表示总是机器可直接识别的地址

C 语言并未定义指针在内存中应该如何表示。理论上讲，指针变量的实际表示可以是任何东西，只要在解引用时它返回的值是按照 C 语言语义指向的变量的值就可以了[1]。这使得在严格的 C 标准下将指针对齐到某个内存粒度是不可能的，因为无法对其使用按位与来掩蔽掉它的某些位。因此，我们假设指针的实际表示总是机器可直接识别的二进制地址，这一点在所有 RMP 支持或计划支持的架构上都是成立的。

7.7.4 C 未定义行为：强制转换指针到不小于机器字长的无符号整数值总是不改变它的实际表示

C 语言并未规定指针在转换到无符号整数后，其实际表示应如何变化。它只规定了，如果先把一个指针转换成一个空间足够大的无符号整数后再转换回去，来回转换后得到的指针在和原指针做比较操作时应当视为相等[1][1]。如果指针在转换到无符号整数后的表示会变化，那么我们会遇到与 7.7.3 所述类似的问题。因此，我们假设强制转换指针到一个机器字长的无符号整数值总是不改变它的实际表示，这一点在所有 RMP 支持或计划支持的架构上都是成立的。

事实上，RMP 并不禁止在转换到更长的无符号整数时在该整数的长于原指针的高位添加任何内容[2]，只要这些高位内容在转换回原指针类型后被丢弃；这是因为，RMP 只会对该无符号整数的最低几位进行掩蔽操作，然后再将该无符号整数转换成指向同一类别存储器空间[3]的（1）原指针类型或（2）不比原指针类型的内存对齐条件更苛刻的指针类型使用，不会涉及到潜在的高位内容。不过，在使用此种移植时必须注意，由指针转换成的无符号整数会具备意外的高字节垃圾。

7.7.5 C 未定义行为：一切“严丝合缝”的结构体会自动紧凑排列

C 语言并未规定结构体中各变量的内存地址对齐方式。理论上讲，编译器可以任意决定结构体中的变量的对齐方式，只要它保证结构体的访问语义就可以了[1]。这甚至不禁止将一个仅含有一种类型的结构体对齐到更大的任意粒度[4]。因此，即便结构体在类型方面是“纯洁”的，我们也无法保证它内部的变量是紧凑排列的。为解决 C 语言规范在这方面的缺失，几乎所有编译器都提供了额外机制来使结构体中各变量紧凑排列[5]，但它们提供的手段堪比巴别塔：有的编译器使用#pragma，有的编译器使用__attribute__，而且不同编译器的机制在结构体声明中的位置是不同的。使用#pragma 的编译器尤其棘手，因为宏无法展开成#pragma；该#pragma 必须被直接插入代码文件中。

为了避免上述问题，RMP 假设编译器总是自动紧凑排列一切“严丝合缝”的结构体，其中“严丝合缝”的定义如下：

---

[1] 这甚至不禁止转换前和来回转换后得到的表示有差异，只要这两个值在做指针比较时总是按照相等处理即可

[2] 如 RL78 的 CC-RL 编译器会在内存指针转换得到的无符号整数的高位插入 0x000F，代表这是指向内存空间

[3] 部分架构的数据和代码存储器是分离的；数据和代码存储器也可能各有多种子类别

[4] 比如将仅含 char 类型的结构体中的每个变量对齐到 1GiB；这种荒唐行为是 C 语义容许的

[5] 其中最出名的是 GCC 的__attribute__((packed))

1. 结构体中的“最终基础类型”的大小都是 2 的次方；其中，“最终基础类型”是指递归组成该结构体类型的编译器原生数据类型；

2. 若结构体被紧凑排列，则其中任何“最终基础类型”的变量的偏移地址都自然对齐到该“最终基础类型”的大小；

3. 结构体的基地址对齐要求不高于其中最长的“最终基础类型”的对齐要求。

几乎所有的编译器都满足上述要求，因为对齐变量的唯一目的是防止发生可能影响性能或造成异常的非对齐访问。有鉴于此，RMP 不保证支持不满足这个要求的编译器。

7.7.6 C 未定义行为：外部标识符的长度扩展为 31 字符

C90 规定外部（extern）变量名最少只有前 6 个字符是有效的。这对任何操作系统来说都太少了。因此，RMP 把这个限制放宽到了与内部变量名限制相同的 31 个字符。所有的现代编译器均支持这一点。

7.7.7 MISRA-C 背离情况说明

RMP 对于 MISRA-C:2012[2]中的所有三类规则都进行了严格检查，并且所有的背离都被批准。

RMP 对于 MISRA-C:2012[2]中所述的强制（Mandatory）类别没有任何背离发生。

RMP 对于 MISRA-C:2012[2]中所述的要求（Required）类别的背离如下：

表 7-4 RMP 对 MISRA-C:2012 中所述的要求（Required）部分的背离

条目	解释
1.3	某些如前所述的未定义行为对于实现操作系统这样的底层软件而言是必需的。
5.1	RMP 扩展了外部标识符的有效长度到 31 位。这对于操作系统的软件工程而言是必需的。
8.4	RMP 使用特殊的头文件编写方法规避了这一问题，并使得变量的声明和定义可以在一起进行。这有利于提高软件质量。
11.3 11.6	类型双关指针别名和指针到足够长度的整数的转换对于操作系统这样的软件而言是必需的。RMP 事实上仅在（1）结构体和子结构体处、（2）HAL 接口处和（3）内存池处使用此类双关指针别名。
11.9	为防止与其它第三方源码包冲突，RMP 不使用 NULL 而使用 RMP_NULL 来表示空指针。
14.3	RMP 在检查配置是否正确的流程中大量使用了编译时就能确定结果的布尔表达式。这些布尔表达式都被直接送入 RMP_ASSERT 宏，确保 RMP 被正确配置。这提高了软件的质量。
17.7	由于 RMP 的硬件抽象层要对所有的受支持硬件定义统一的接口，因此不免会有一些硬件抽象层或/和调试打印函数的返回值对于某些架构而言是多余的。
21.1	RMP 声明了__HDR_DEF__、__HDR_STRUCT__和__HDR_PUBLIC__三个保留宏，它们仅做自用。因此，这并不违反#undef 系统中其他保留宏的规定。另见 20.5 的说明。

RMP 对于 MISRA-C:2012[2]中所述建议（Advisory）类别的背离如下：

表 7-5 RMP 对 MISRA-C:2012 中所述的建议（Advisory）部分的背离

条目	解释
2.3 2.5	在硬件抽象层，尤其是固件库中，某些宏或数据类型是打包一起提供的。它们可能在当前配置中没有使用，但必须保留以适应未来的软件配置调整，有利于维护工程的一致性和提高代码质量。这些宏都有很好的注释和文档可供参考。
2.7 8.7	由于 RMP 的硬件抽象层要对所有的受支持硬件定义统一的接口，因此不免会有一些硬件抽象层函数的参数对于某些架构而言是多余的，或者部分声明了 extern 的函数未被实际调用，等等。
8.9	RMP 有部分全局变量仅被一个函数引用，但并不放在该函数内部，这是为了方便用户进行代码调试。
11.4 11.5	转换一个无符号数或者 void*到常规指针对于操作系统这样的软件而言是必需的。具体解释见 11.3 的相关说明。
15.5	RMP 的一个函数可能会有多个出口。在这些代码处都进行了良好注释，并且其他可选实现会违反其他的要求或建议，并且破坏工程的整体抽象。
20.5	为了使每个函数和变量的声明都只进行一次（包括其 extern 形式），系统性地防止出现隐式声明错误来提高软件质量，RMP 内部使用了特殊的分段包含式头文件。该格式要求使用#undef。#undef 仅用于该内部格式，不会暴露到内核外部，且该格式在整个系统中都被良好遵循，并有对应文档进行详细说明。

7.8 本章参考文献

[1] ISO, International Standards Organisation, "Programming Languages—C, International standard, ISO/IEC 9899: 1990 (E)", 1990.

[2] M. I. S. R. Association, MISRA C 2012: Guidelines for the Use of the C Language in Critical Systems: March 2013: Motor Industry Research Association, 2013.