RMP Light-Weight RTOS
TRM

Rev. R4T5

演进·远古·原核（五阶）

Mutatus·Protero·Prokaron(R.V)

M5P1(Prokaron) R4T5

Light-Weight RTOS (Rev.4 Typ. 5)

Technical Reference Manual

System Features

1.High Usablility

- Light-weight and ease of use as first-class requirements
- Easy to configure development environment and project path
- Provides generic inter-thread communication interfaces

2.Hard Real-Time

- Fully preemptive hard-real-time kernel
- Round-robin scheduling for threads with the same priority
- O(1) time complexity for all key kernel operations

3.High Portability

- Least assembly required amongst all RTOSes
- Strict ISO/ANSI C89 & MISRA C compliance ensures full compatibility
- Least resource required amongst all RTOSes
- Can run as a guest OS on embedded hypervisors

4.High Efficiency

- Highly optimized performance-critical routines
- Lightening-fast context switching & extremely low interrupt latency

5.High Reliability

- Provides argument assertion and pervasive parameter checking
- Formally verified to reach IEC 61508 SIL4 or Orange book A1 or EAL7+

  *Work in progress; current software is SIL2 precertified equivalent

Contents

List of Tables

List of Figures

Revision History

Version	Date (YYYY-MM-DD)	Notes
R1T1	2018-02-15	Initial release
R2T1	2018-03-12	Add porting guide and appendix
R4T3	2018-07-31	Fix manual errors
R4T3	2018-09-01	Update the manual to new version
R4T4	2018-10-10	Lift 128MB restriction on memory manager and optimize rounded rectangle drawing algorithm
R4T5	2018-11-25	Lift the restriction on number of priorities and allow further footprint reduction

Chapter 1 Introduction

1.1 Preface

In miniature IoT systems, 16-bit and 32-bit processors are becoming increasingly prevalent. In the meantime, the concerns for low development time and high system reliability is rising. Thus, it is necessary to develop new light-weight Real-Time Operating Systems (RTOSes) for simple IoT application implementations. With the advent of light-weight virtualization in modern embedded systems, light-weight OS may also be used as a guest OS to run on embedded hypervisors. Moreover, there's no memory protection for such operating systems, thus formal verification are necessary to reach high reliability.

RMP is a fully-preemptible RTOS highlighting simplicity, usability and efficiency. It implemented a fixed-priority round robin scheduler, along with many features commonly found in many other RTOSes. It also features a simple memory management interface as a standalone module, some additional GUI interface support, and the ability to exploit hardware features for execution speed. To sum up, RMP can be deployed on microcontrollers (MCUs) with as little as 2kB ROM and 1kB RAM.

This manual described the APIs of RMP kernel from the user's perspective. We will review some basic concepts of RTOSes before we detail RMP and its APIs.

1.1.1 Design Goal and Specifications

The first-class design goal is to make RMP open-source, as simple and usable as possible, while being reliable and real-time at the same time. In addition to this, it must outclass other RTOSes in terms of resource consumption and average efficiency.

1.1.2 Copyright Notice and License

Taking the license requirements of different applications into consideration, RMP adopted LGPLv3 as its main license. For some special cases[1], some special terms apply. These special terms might be different for each particular application.

1.1.3 Terms and Definitions

The terms and abbreviations used in this manual are listed as follows:

[1] E.g. security and medical equipment

1.1.3.1 Operating System

The lowest level software infrastructure which is responsible for processor, memory and device management.

1.1.3.2 Thread

A control flow that have one standalone stack and can be scheduled independent of each other. There can be multiple threads in one process address space.

1.1.3.3 Static Allocation

All system resource allocations are done at compile time.

1.1.3.4 Dynamic Allocation

At least a part of the resources can be allocated or freed at runtime.

1.1.3.5 Soft Real-time

A system that meets most of its deadline requirements. Some deadline misses are acceptable, provided that these cases are rare.

1.1.3.6 Hard Real-time

A system that meets all of its deadline requirements. Any deadline misses are not allowed.

1.1.3.7 Constant Real-time

All operations are O(1) with regards to user input and system configuration. The constant time factor must be reasonable and small enough. This is the strongest real-time guarantee.

1.1.3.8 Constant Real-time to a Certain Value

All operations are O(1) when the value is given, and the constant factor must be reasonable and small enough.

1.1.4 Major Reference Systems

Scheduler : FreeRTOS (@RT Engineering LTD/Amazon).

Formal proof : seL4 (@2016 Data61/CSIRO).

API : RMProkaron (@EDI).

All other references will be listed in respective chapters.

1.2 Forewords

Operating system is a category of basic software that is responsible for low-level CPU, memory and device management. For RTOSes, all operations of the system must be predictable and always meet its deadline. Generally speaking, there are two subtypes of real-time systems: soft real-time systems, which meets its deadline in most cases, and hard real-time systems, which meets its deadline in all cases. Practically no system is completely hard real-time or soft real-time; all of them can be split in half, one portion being soft real-time and the other portion being hard real-time. One example is the motor controller: the LCD GUI part is soft real-time, an the motor controller part is hard real-time.

RMP belongs to the basic hard real-time system class, which exhibits all the fundamental features of a RTOS. This class of systems should be deployed on high-end 8/16-bit and 32-bit machines at most times, and requires a system tick timer. These systems does not have genuine kernel space and user space separation; however, it is possible to configure the MPU or MMU to protect some ranges of memory. The hardware abstraction layers of these systems include some simple assembly, which must be modified when porting to other architectures. The porting also involves system tick timer, context switching, interrupt management and coprocessor management. These systems can use a customized linker script; however this is not always necessary, except in the case of memory protection.

In these systems, a task is always a thread, and can be reentrant. Threads have their own respective stacks. The application code can be either linked with the kernel or as standalone modules. There are no system calls and the system API is just normal function calls.

These systems provide priorities, and the threads on the same priority level are scheduled with round-robin algorithm, while threads on different priorities will preempt each other. They also have primitive memory management support, which is usually based on SLAB or buddy system.

Interrupts can be totally transparent to the system, and when so the system will be totally unaware of the interrupts. When it a context switch is needed in an interrupt, we must insert context switching stubs into the interrupt routine, and some assembly programming is required.

Typical such systems include RT-Thread, FreeRTOS, uC/OS, Salvo and ChibiOS. It is usually easy to perform a complete formal verification on such systems, due to the minimal number of lines of kernel code. Contrary to traditional software engineering which tests the functionality of the software systematically, formal approaches seek to apply formal logic and eventually prove that the system is free of bugs under given preconditions.

1.3 Performance and Specifications of RTOSes

There are hundreds of different RTOSes out there on the market, and kernels developed by hobbyists are virtually uncountable. The performance and reliability of these systems vary, and

we need some measures to benchmark them. All the measurements listed below can only be directly compared against each other when the processor architecture, compiler, and compiling options are all the same. If different processors, compilers or compiler options are used, then the results cannot be directly compared to each other and can only serve as a reference. One recommended approach is to use industry-standard ARM or MIPS processors and GCC -O2 compilation option. Simulators such as Chronos are also useful. When performing the evaluation, system load will also influence the results, thus the system load must be the same when comparing the results.

1.3.1 Kernel Size

Kernel size is a key aspect of RTOSes. RTOSes are usually deployed on resource-constrained devices, thus a small kernel size is critical. The size of the kernel will be evaluated in two dimensions, respectively being the read-only memory size and read-write memory size. The read-only size includes the code and constant data segment, while the read-write size includes the modifiable data segment. On Flash-based MCUs, the read-only segments will consume Flash, and the read-write segments will consume SRAM[1].

RTOSes are highly configurable, thus their kernel size is rarely a fixed number and these numbers are tied closely to a specific configuration. Therefore, to measure this performance, you should measure the kernel size under the minimal kernel configuration, common kernel configuration, and full-featured kernel configuration[1].

Obtaining kernel size data is as simple as compiling the kernel with a compiler and then using a dedicated binary viewer[1] to inspect the size of each section of the target file.

1.3.2 Execution Time, Worst-Case Execution Time and Jitter

The execution time refers to the time consumption of a particular RTOS system call. The Worst Case Execution Time (WCET) refers to the maximum length of execution time under the most unfavorable conditions. WCETs is usually achieved when the most tedious system call is made and the most number of cache misses occur. RTOSes generally disable interrupts when executing of system calls; the worst execution time is usually the longest interval in which the system disables interrupts, so the impact of WCETs on the real-time performance is enormous.

These WCETs can be divided into two categories: the first is the WCETs of system calls, and the second is the WCETs of inter-thread synchronizations.

To get the first type of WCETs, before calling a system call, jot down the time stamp T_s at this time, and then after the end of the system call, read the timer to get the time stamp T_e. Then,

[1] Such as Objdump

read the timer twice in a row to get two timestamps, T_{ts} and T_{te}, and obtain the extra cost of reading the timer as T_{te}-T_{ts}. In this case, the execution time is T_e-T_s-(T_{te}-T_{ts}). Repeat this on all system calls, then the WCET will be largest number among all measurements.

To get the second type of WCETs, jot down the timestamp T_s at the sending side of the communication channel, then at the receiving end, read the timer get the timestamp T_e. The measurement for the cost of reading the timer is similar to the first-type WCETs. The resulting T_e-T_s-(T_{te}-T_{ts}) is the execution time. Repeat this on all communication mechanisms calls, then the WCET will be largest number among all measurements.

Jitter of execution time is also very important. We often get a distribution when conduct the same measurement multiple times. The average of this distribution is the execution time, and its standard deviation[1] is called the jitter.

For a RTOS, we usually wish the execution time, the WCET and the jitter to be as small as possible. Execution time can be further divided into the following categories in detail[1]:

1.3.2.1 Thread Context Switch Time

Thread context switch time is the time cost of switching from one thread to another. We use the following method to measure this value. In the measurement, except for the method of using T_e-T_s, it is also possible to use the difference between two consecutive T_s divided by 2. There are two types of context switching, one is to switch between threads with the same priority, and the second is to wake up a high-priority thread in low-priority thread[2].

In the first case, we assume that the two threads have the same priority, and at the beginning of the measurement we are executing thread A, which was just switched to by thread B.

Table 1-1 The First Case of Thread Context Switch

Thread A	Thread B
Loop forever	Loop forever
{	{
>> Read T_s;	Read T_e;
Switch to thread B;	>> Switch to thread A;
}	}

In the second case, we assume that the thread B have a higher priority, and at the beginning of the measurement we are executing the thread A, which was just switched to by thread B.

[1] In some cases we also use range

Table 1-2 The Second Case of Thread Context Switch

Thread A	Thread A
Loop forever	Loop forever
{	{
>> Read T_s;	Read T_e;
Wake B up;	>> Sleep forever;
}	}

1.3.2.2 Asynchronous Communication Time

Asynchronous communication time refers to the time to communicate between two threads with asynchronous channels. We use the following method to measure this value. We assume that thread B is blocked at the receiving end, thread A sends to thread B, and thread B has a higher priority than thread A.[2]

Table 1-3 Asynchronous Communication between Threads

Thread A	Thread B
Loop forever	Loop forever
{	{
>> Read T_s;	Read T_e;
Send to thread B;	>> Receive from own mailbox;
}	}

1.3.3 Interrupt Response Time, Worst-Case IRT and Jitter

Interrupt response time refers to the time between the occurrence of an interrupt and the corresponding processing thread's wakeup. The Worst-Case Interrupt Response Time (WCIRT) refers to the maximum length that an interrupt response time can reach under the most unfavorable conditions. The WCIRT is usually reached when a large number of cache misses and Trans Look-aside Buffer (TLB) misses occur during interrupt processing. Interrupt response time is the most important indicator of the RTOS, and it can even be asserted that everything of the RTOS should be designed around it. This measurement is the most direct reflection of the RTOS's real-time performance.

To obtain the WCIRT, a timer can be read in the first line of assembly of the interrupt vector[1], resulting in a time stamp T_s; read the timer at the first line of code of the interrupt processing thread to get a timestamp T_e. The measurement of the cost of readinf the timer is the same as above. The resulting T_e-T_s-(T_{te}-T_{ts}) is the interrupt response time. The worst-case interrupt response time is the one with the highest response time among all the interrupt-response tests.

The jitter of this is also critical. We often get a distribution when we measure the IRT of the same system multiple times. The average of this distribution is the average interrupt response time, and its standard deviation[2] is called interrupt response time jitter.

For a RTOS, we usually wish the IRT, the WCIRT and the jitter to be as small as possible. Interrupt response time measurement is usually performed as follows [1][3]:

Table 1-4 Interrupt Response Time

Kernel	Thread A
Hardware interrupt handler	Hardware interrupt handler
{	{
>> Read T_s;	Read T_e;
Send to thread A from interrupt;	>> Receive from own mailbox;
}	}

1.3.4 Realistic IRT, Realistic Worst-Case IRT and Jitter

The realistic interrupt response time refers to the time between the external stimulus's input and the corresponding I/O operations' completion. The realistic worst-case interrupt response time refers to the maximum length that an realistic interrupt response time can reach under the most unfavorable conditions. In addition to the factors that can affect the WCIRT, CPU and IO hardware's inherent overhead will also affect realistic WCIRT.

To get realistic IRT, we will need some extra hardware to conduct the measurement. For example, we need to measure the actual interrupt response time of an I/O line. We can connect the output pin of a FPGA to the input pin of a CPU or motherboard, then connect the input pin of the same FPGA to the CPU or motherboard's output pin. Firstly, the FPGA sends a signal on its output pin. At this time, the high-resolution timer in the FPGA starts counting. When the FPGA receives the signal on its input pin, the high-resolution timer in the FPGA stops counting. The

[1] We cannot wait until the C handler starts execution because register and stack maintenance are also part of the interrupt response time

[2] In some cases we also use range

resulting internal FPGA timer value is the realistic IRT. The realistic WCIRT is the one with the highest response time among all tests.

For a cyber-physical system, we wish that the realistic IRT, the realistic WCIRT and the jitter are as small as possible. It is noteworthy that the realistic WCIRT will generally be approximately equal to the WCET plus the WCIRT plus the hardware's inherent overhead. For example, when a system is just starting to execute a system call at the time of stimulus, the hardware interrupt vector cannot be executed immediately, and the system call must be completed before we can respond to it. After the system call is completed, the hardware interrupt vector begins to execute, after this we will switch to the processing thread and produce the output. The realistic IRT is usually measured as follows[1]:

Table 1-5 Realistic Interrupt Response Time

FPGA (or oscilloscope)	System under test
Loop forever	Loop forever
{	{
>> Send stimulus and start the timer;	Receive signal from I/O;
Wait until receipt of the response;	Minimal processing routine;
Stop the timer;	Send response to I/O;
}	}

1.4 RMP System Calls

System call is the only way to use the system's functionality. For RMP, the system calls are actually identical to regular function calls. All system calls in RMP are listed hereinafter.

Table 1-6 List of RMP System Calls

System call name	ID	Explanation
RMP_Yield	0	Yields the processor of the current thread
RMP_Thd_Crt	1	Create a thread
RMP_Thd_Del	2	Delete a thread
RMP_Thd_Set	3	Set a thread's priority and timeslice
RMP_Thd_Suspend	4	Pend a thread
RMP_Thd_Resume	5	Resume a thread from pend
RMP_Thd_Delay	6	Delay for a interval
RMP_Thd_Cancel	7	Cancel the delay of a thread

System call name	ID	Explanation
RMP_Thd_Snd	8	Send to a thread's mailbox.
RMP_Thd_Snd_ISR	9	Send to a thread's mailbox from interrupt
RMP_Thd_Rcv	10	Receive a value from own mailbox
RMP_Sem_Crt	11	Create a counting semaphore
RMP_Sem_Del	12	Delete a counting semaphore
RMP_Sem_Pend	13	Try to pend for a semaphore
RMP_Sem_Abort	14	Cancel the thread's pend on semaphore
RMP_Sem_Post	15	Post to a semaphore
RMP_Sem_Post_ISR	16	Post from interrupt to a semaphore

1.5 Bibliography

[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.

Chapter 2 System Kernel

2.1 Kernel Introduction

The RMP kernel provided all the basics found on a usable commodity RTOS, including threads, thread delays, thread mailboxes and counting semaphores. More complicated communication facilities can be easily developed with existing built-in facilities of RMP. This system also featured a light-weight Graphics User Interface (GUI) library, which does not consume any memory when not enabled[1].

RMP is recommended for systems where less than 32 priorities and 64 threads are present. Theoretically, an infinite number priorities and threads can be supported; however, from a usability standpoint, it is not recommended to implement such systems with RMP. If more sophisticated embedded system implementations are required, the higher-ranked RMEukaryon (M7M1) is more appropriate.

2.1.1 Scheduler

RMP features a fixed priority round robin scheduler. It employs preemptive scheduling between threads with different priorities, and round-robin scheduling between threads with the same priority. Different from some RTOSes, the RMP scheduler never disables interrupts on some architectures, which guarantees the real-time properties of the system. To further boost the real-time performance, when the CLZ instruction is available on the processor, RMP will employ it to speed to priority searches.

RMP doesn't have built-in multi-core support[2]. However, it is possible to naively support multi-core by running a different instance of RMP on each core, which resembles Barrelfish's[1] separation kernel approach.

2.1.2 Memory Management and Memory Protection

RMP provided a memory allocator based on the Two-Level-Segregated-Fit (TLSF)[2][3] approach. This allocator is O(1) in all allocation and free operations, and have a very high space efficiency. However, using dynamic memory allocation in small embedded systems is generally not recommended. RMP does not feature genuine memory protection, and there is no sepration of address space between threads. Thus, it is mandatory that some programming standard be adhered to when writing applications.

---

[1] No relevant macros are defined

[2] And will never in the future as well

2.1.3 Application Updating and Application Modules

As RMP is a small system, no application updates and application modules are inherently supported. Usually, the resulting image is less than 128kB and can be updated on the whole. If such functionality is desired, it can be implemented by the user manually.

2.1.4 System Booting Sequence

The first thread of RMP is always RMP_Init, which is in the kernel code provided. Init is responsible for boot-time initialization and low-power mode management, and shall never stop running. Two hooks RMP_Init_Hook and RMP_Init_Idle are provided, and they are responsible for importing user code to the booting process. Please refer to section 2.5 for detailed description.

2.2 Thread-Related Operations

Because RMP-based systems run all their applications in the same address space, RMP only provides threads as the abstraction for control flow. There can be four states for each thread, namely READY, RUN, PEND and BLOCK[1]. The BLOCK state can be set to return upon timeout if the requested service is unavailable. The PEND state can be imposed on any thread, upon which the thread will always stop its execution no matter what state it was in.

The complete thread state transition graph is included hereinafter:

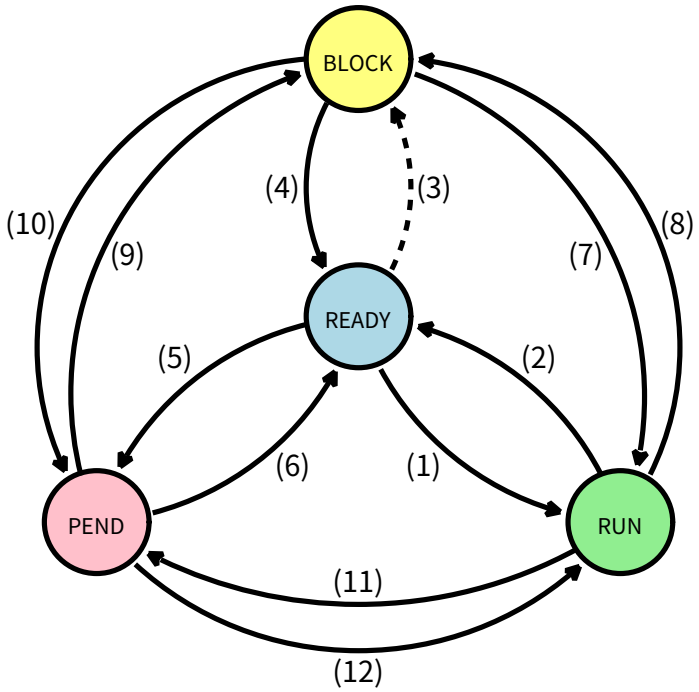

Figure 2-1 Thread State Transition Graph

[1] Includes four variants: blocked on delay or semaphore or send or receive

The meanings of the number markings are as follows:

Table 2-1 Meanings of the Number Markings in the Thread State Transition Graph

No.	Meaning
(1)	The thread have the highest priority and is READY, will go from READY to RUN.
(2)	The thread is in RUN and does not have the highest priority, will go from RUN to READY.
(3)	There is no such possibility, a thread cannot directly enter BLOCK from READY (this is illustrated by a dotted line in the graph).
(4)	The thread is unblocked and not pended, but does not have the highest priority, will go from BLOCK to READY.
(5)	The thread is pended while in READY, will go from READY to PEND.
(6)	The thread is unpended and not blocked, but does not have the highest priority, will go from PEND to READY.
(7)	The thread is unblocked and not pended, and have the highest priority, will go from PEND to RUN.
(8)	The thread is blocked on mailbox, semaphore or delay, will go from RUN to BLOCK.
(9)	The thread is unpended but still in block, will go from PEND to BLOCK.
(10)	The thread is pended while in BLOCK, will go from BLOCK to PEND.
(11)	The thread is pended while in RUN, will go from RUN to PEND.
(12)	The thread is unpended and not blocked, and have the highest priority, will go from PEND to RUN.

All thread related interface are listed below and they can only be called in normal threads; calling them in interrupts is prohibited. They can have the following return values:

Table 2-2 Possible Return Values for Thread Related Interfaces

Return value	Value	Meaning
RMP_ERR_THD	-1	Operation failed due to thread control block relayed issues.
RMP_ERR_PRIO	-2	Operation failed due to priority related issues.
RMP_ERR_SLICE	-3	Operation failed due to timeslice related issues.
RMP_ERR_STATE	-4	Operation failed due to thread state related issues.
RMP_ERR_OPER	-5	Operation failed due to other issues.
RMP_ERR_SEM	-6	Operation failed due to semaphore issues.

All system calls and their return values are listed hereinafter.

2.2.1 Yield to Another Thread

This operation will cause the current thread to give up CPU, and the scheduler will automatically pick the next thread to schedule. If the current thread is the only thread with the highest priority, then the same thread will still be scheduled.

Table 2-3 Yield to Another Thread

Prototype	void RMP_Yield(void)
Return	None.
Parameter	None.

2.2.2 Thread Creation

This operation will create a new thread and put it into READY state. RMP does not provide kernel object management facilities thus the thread control block needs to be allocated by the user. The Stack parameter of this function is the start address of the thread stack; for ascending stacks, this address is at the low-end of the stack memory; for descending stacks, this address is at the high-end of the stack memory.

It is worth noting that the stack address passed in will be directly assigned to the current stack pointer of the thread, and on the first context restoration of the thread, the stack popping will begin here. Thus it is necessary to reserve some space at the top of the stack for context initialization in thread creation. If other information needs to be stored on the stack, this space also needs to be reserved.

For example, if the size of Stack_Array is 256 words, the processor has an context size of 16 words and uses a descending stack, and we wish to reserve 10 extra words for safety redundancy. Then, the Stack parameter passed to this function should be &Stack_Array[256-16-10]. In this example, if the processor uses a ascending stack, then the Stack parameter should be &Stack_Array[16+10].

Table 2-4 Thread Creation

Prototype	rmp_ret_t RMP_Thd_Crt(volatile struct RMP_Thd* Thread, rmp_ptr_t Entry, rmp_ptr_t Stack, rmp_ptr_t Arg, rmp_ptr_t Prio, rmp_ptr_t Slices)
rmp_ret_t	

Return	If successful, 0. If failed, one the following values will be returned:	
	RMP_ERR_PRIO	The priority is not smaller than RMP_MAX_PREEMPT_PRIO.
	RMP_ERR_SLICE	The timeslice is 0 or not smaller than RMP_MAX_SLICES.
	RMP_ERR_THD	The thread control block is 0 (NULL) or being used.
Parameter	volatile struct RMP_Thd* Thread A pointer to an empty thread control block to be used for this thread.	
	rmp_ptr_t Entry The entry of the thread.	
	rmp_ptr_t Stack The execution stack of the thread. For architectures that use a grow-down stack, it points to the high end of the stack space; for architectures that use a grow-down stack, it points to the low end of the stack space.	
	rmp_ptr_t Arg The argument to pass to the thread.	
	rmp_ptr_t Prio The priority of the thread.	
	rmp_ptr_t Slices The number of timeslices for the thread. The unit of timeslice is ticks.	

2.2.3 Thread Deletion

This operation will delete a thread in the system. When the thread is deleted, all its resource will be revoked. If there are any thread sending to it, the send will be canceled and a failure will be returned.

It is worth noting that RMP threads does not allow exiting by a direct return; if it wants to terminate itself, it must call this function to delete itself. This restriction is designed to make thread deletion explicit to the user, thus deprecating the “create - run once - delete” programming style[1].

Table 2-5 Thread Deletion

Prototype	rmp_ret_t RMP_Thd_Del(volatile struct RMP_Thd* Thread)
Return	rmp_ret_t If successful, 0. If failed, one the following values will be returned:

[1] Such style should be replaced by a “block - run once - block again” style

	RMP_ERR_THD	The thread control block is 0 (NULL) or being used.
Parameter	volatile struct RMP_Thd* Thread A pointer to the thread control block of the thread to be deleted.	

2.2.4 Setting Thread Parameters

This operation will set a thread's timeslice and priority. When setting only one of them is required, it is necessary to keep the other parameter the same with its original value.

Table 2-6 Setting Thread Parameters

Prototype	rmp_ret_t RMP_Thd_Set(volatile struct RMP_Thd* Thread, rmp_ptr_t Prio, rmp_ptr_t Slices)	
Return	rmp_ret_t If successful, 0. If failed, one the following values will be returned:	
Return	RMP_ERR_PRIO	The priority is not smaller than RMP_MAX_PREEMPT_PRIO.
	RMP_ERR_SLICE	The timeslice is 0 or not smaller than RMP_MAX_SLICES.
	RMP_ERR_THD	The thread control block is 0 (NULL) or not used.
Parameter	volatile struct RMP_Thd* Thread A pointer to the thread control block of the thread to be set parameters.	
	rmp_ptr_t Prio The priority to set to the thread.	
	rmp_ptr_t Slices The timeslice to set to the thread.	

2.2.5 Thread Suspension

This operation will suspend a thread and force it into the PEND state. Whatever state the thread is in, it can be suspended, however its former state will be retained in some sense. If the thread was in delay or send or receive before it gets suspended, these states are still valid in the background and these operations will still proceed. If these operations are completed while the thread is suspended, then the thread will stay in a pure PEND state and will not be scheduled until the PEND state is manually resumed.

Table 2-7 Thread Suspension

Prototype	rmp_ret_t RMP_Thd_Suspend(volatile struct RMP_Thd* Thread)
Return	rmp_ret_t

	If successful, 0. If failed, one the following values will be returned:	
	RMP_ERR_THD	The thread control block is 0 (NULL) or not used.
	RMP_ERR_STATE	The thread is already PEND and cannot be suspended again.
Parameter	struct RMP_Thd* Thread A pointer to the thread control block of the thread to be suspended.	

2.2.6 Thread Resumption

This operation will resume the suspended state of a thread, and make it schedulable again.

Table 2-8 Thread Resumption

Prototype	rmp_ret_t RMP_Thd_Resume(volatile struct RMP_Thd* Thread)	
Return	rmp_ret_t If successful, 0. If failed, one the following values will be returned:	
	RMP_ERR_THD	The thread control block is 0 (NULL) or not used.
	RMP_ERR_STATE	The thread is not suspended and cannot be resumed.
Parameter	volatile struct RMP_Thd* Thread A pointer to the thread control block of the thread to be resumed.	

2.2.7 Thread Delaying

This operation will delay the current thread for some time.

Table 2-9 Thread Delaying

Prototype	rmp_ret_t RMP_Thd_Delay(rmp_ptr_t Slices)	
Return	rmp_ret_t If successful, 0. If failed, one the following values will be returned:	
	RMP_ERR_SLICE	The timeslice is 0 or not smaller than RMP_MAX_SLICES.
	RMP_ERR_OPER	The delay was cancelled before it completed.
Parameter	rmp_ptr_t Slices The number of timeslices to delay.	

2.2.8 Thread Delay Cancellation

This operation will cancel a thread's delay operation.

Table 2-10 Thread Delay Cancellation

Prototype	rmp_ret_t RMP_Thd_Cancel(volatile struct RMP_Thd* Thread)	
Return	rmp_ret_t	
	If successful, 0. If failed, one the following values will be returned:	
	RMP_ERR_THD	The thread control block is 0 (NULL) or not used.
	RMP_ERR_STATE	The thread is not in delay state and cannot be canceled.
Parameter	volatile struct RMP_Thd* Thread	
	A pointer to the thread control block of the thread to be canceled.	

2.3 Thread Communication Interfaces

RMP provides a relatively simple but efficient mechanism for inter-thread communication, including thread mailboxes and semaphores. These mechanisms can be used alone or combined to construct more complex communication interfaces. Blockable communication interface has three options: return immediately upon detection of possible blocking, return after the timeout or block indefinitely. This greatly improves the flexibility of the interface. Note that for all communication interfaces, when a number of threads are blocked on one of them, the service is always first-come first-served, and no queue cutting functionality is provided. If a high-priority thread is blocked after a low-priority thread have blocked, the first one to be serviced is the low-priority thread and then the high-priority thread.

2.3.1 Sending to Thread Mailboxes

This operation will send a message of one processor word to a thread's mailbox.

Table 2-11 Sending to Thread Mailboxes

Prototype	rmp_ret_t RMP_Thd_Snd(volatile struct RMP_Thd* Thread, rmp_ptr_t Data, rmp_ptr_t Slices)	
Return	rmp_ret_t	
	If successful, 0. If failed, one the following values will be returned:	
	RMP_ERR_THD	The thread control block is 0 (NULL) or not used.
	RMP_ERR_OPER	Possible block detected when a non-blocking option is specified, or trying to send to its own mailbox, or the send failed because of timeout, or the target thread is deleted.
Parameter	volatile struct RMP_Thd* Thread	
	A pointer to the thread control block of the thread to send to.	

	rmp_ptr_t Data The data to send.
	rmp_ptr_t Slices The number of timeslices to wait in case a block may happen. If 0 is specified, upon detection of possible blocking, the function will return immediately. If a value between 0 and RMP_MAX_SLICES is specified (RMP_MAX_SLICES is not included), then the operation will block for at most this value of timeslices. If a value greater or equal to RMP_MAX_SLICES is specified, then the operation will indefinitely block until the send operation is complete or the target thread gets destroyed.

2.3.2 Sending to Thread Mailboxes from Interrupts

This operation will send to a thread from an interrupt handler. Different from the version described above, this version will always return without blocking. This function will not lock the scheduler, and does not care whether the scheduler is locked or not. This is because if the interrupts where this function can be called is entered, the scheduler is definitely not locked.

Table 2-12 Sending to Thread Mailboxes from Interrupts

Prototype	rmp_ret_t RMP_Thd_Snd_ISR(volatile struct RMP_Thd* Thread, rmp_ptr_t Data)	
Return	rmp_ret_t If successful, 0. If failed, one the following values will be returned:	
	RMP_ERR_THD	The thread control block is 0 (NULL) or not used.
	RMP_ERR_OPER	The target thread's mailbox is full.
Parameter	volatile struct RMP_Thd* Thread A pointer to the thread control block of the thread to send to.	
	rmp_ptr_t Data The data to send.	

2.3.3 Receiving from Thread Mailboxes

This operation receive a value from the thread's mailbox.

Table 2-13 Receiving from Thread Mailboxes

Prototype	rmp_ret_t RMP_Thd_Rcv(rmp_ptr_t* Data, rmp_ptr_t Slices)
Return	rmp_ret_t If successful, 0. If failed, one the following values will be returned:

	RMP_ERR_OPER	Possible block detected when a non-blocking option is specified, or the receive failed because of timeout, or a null pointer is passed in.
Parameter	rmp_ptr_t* Data This parameter is used for output, outputs the data received.	
	rmp_ptr_t Slices The number of timeslices to wait in case a block may happen. If 0 is specified, upon detection of possible blocking, the function will return immediately. If a value between 0 and RMP_MAX_SLICES is specified[1], then the operation will block for at most this value of timeslices. If a value greater or equal to RMP_MAX_SLICES is specified, then the operation will indefinitely block until the receive operation is complete.	

2.3.4 Semaphore Creation

This operation will create a new counting semaphore.

Table 2-14 Semaphore Creation

Prototype	rmp_ret_t RMP_Sem_Crt(volatile struct RMP_Sem* Semaphore, rmp_ptr_t Number)	
Return	rmp_ret_t If successful, 0. If failed, one the following values will be returned:	
	RMP_ERR_SEM	The semaphore control block is 0 (NULL) or used.
	RMP_ERR_OPER	The initial semaphore number is greater than or equal to RMP_SEM_MAX_NUM.
Parameter	volatile struct RMP_Sem* Semaphore A pointer to an empty semaphore control block to be used for this semaphore.	
	rmp_ptr_t Number The initial semaphore number, must be smaller than RMP_SEM_MAX_NUM.	

2.3.5 Semaphore Deletion

This operation will delete a semaphore. If there are threads that are blocked on it, then these threads' pend call will return RMP_ERR_OPER.

[1] RMP_MAX_SLICES is not included

Table 2-15 Semaphore Deletion

Prototype	rmp_ret_t RMP_Sem_Del(volatile struct RMP_Sem* Semaphore)
Return	rmp_ret_t If successful, 0. If failed, one the following values will be returned: RMP_ERR_SEM The semaphore control block is 0 (NULL) or not used.
Parameter	volatile struct RMP_Sem* Semaphore A pointer to the semaphore control block of the semaphore to delete.

2.3.6 Pending for Semaphores

This operation will pend the current thread on a semaphore. The number to pend for is always 1.

Table 2-16 Pending for Semaphores

Prototype	rmp_ret_t RMP_Sem_Pend(volatile struct RMP_Sem* Semaphore, rmp_ptr_t Slices)
Return	rmp_ret_t If successful, 0. If failed, one the following values will be returned: RMP_ERR_SEM The semaphore control block is 0 (NULL) or not used. RMP_ERR_OPER Possible block detected when a non-blocking option is specified, or the pend failed because of timeout, or the semaphore is deleted.
Parameter	volatile struct RMP_Sem* Semaphore A pointer to the semaphore control block of the semaphore to pend for. rmp_ptr_t Slices The number of timeslices to wait in case a block may happen. If 0 is specified, upon detection of possible blocking, the function will return immediately. If a value between 0 and RMP_MAX_SLICES is specified[1], then the operation will block for at most this value of timeslices. If a value greater or equal to RMP_MAX_SLICES is specified, then the operation will indefinitely block until the pend operation is complete, or the semaphore is deleted.

2.3.7 Semaphore Pend Abortion

[1] RMP_MAX_SLICES is not included

This operation aborts a thread's pend on a semaphore. If the pend is successfully aborted, the waiting thread's pend call will return RMP_ERR_OPER.

Table 2-17 Semaphore Pend Abortion

Prototype	rmp_ret_t RMP_Sem_Abort(volatile struct RMP_Thd* Thread)	
Return	rmp_ret_t If successful, 0. If failed, one the following values will be returned:	
	RMP_ERR_THD	The thread control block is 0 (NULL) or not used.
	RMP_ERR_STATE	The thread is not pending on a semaphore and cannot be aborted.
Parameter	volatile struct RMP_Thd* Thread A pointer to the thread control block of the thread to cancel the semaphore pend.	

2.3.8 Posting to Semaphores

This operation will post a certain number of tokens to a semaphore.

Table 2-18 Posting to Semaphores

Prototype	rmp_ret_t RMP_Sem_Post(volatile struct RMP_Sem* Semaphore, rmp_ptr_t Number)	
Return	rmp_ret_t If successful, 0. If failed, one the following values will be returned:	
	RMP_ERR_SEM	The semaphore control block is 0 (NULL) or not used.
	RMP_ERR_OPER	If this number is posted to the semaphore, the semaphore count will exceed RMP_SEM_MAX_NUM.
Parameter	volatile struct RMP_Sem* Semaphore A pointer to the semaphore control block of the semaphore to post to.	
	rmp_ptr_t Number The number of tokens to post.	

2.3.9 Posting to Semaphores from Interrupts

This operation will post a certain number of tokens to a semaphore from an interrupt handler. This function will not lock the scheduler, and does not care whether the scheduler is locked or not. This is because if the interrupts where this function can be called is entered, the scheduler is definitely not locked.

Table 2-19 Posting to Semaphores from Interrupts

Prototype	rmp_ret_t RMP_Sem_Post_ISR(volatile struct RMP_Sem* Semaphore, rmp_ptr_t Number)	
Return	rmp_ret_t If successful, 0. If failed, one the following values will be returned:	
	RMP_ERR_SEM	The signal control block is 0 (NULL) or not used.
	RMP_ERR_OPER	If this number is posted to the semaphore, the semaphore count will exceed RMP_SEM_MAX_NUM.
Parameter	volatile struct RMP_Sem* Semaphore A pointer to the semaphore control block of the semaphore to post to.	
	rmp_ptr_t Number The number of tokens to post.	

2.4 Memory Management Interfaces

RMP provided a TLSF-based memory allocator, which has been proven for dynamic memory management. TLSF is a two-level fit algorithm which separates the memory blocks into different First-Level Intervals (FLIs) according to their power of 2, such as 127-64 Byte, 255-128 Byte and so on. Then it cuts each FLI into smaller Second-Level Intervals (SLIs) linearly. For example, the 127-64 Byte FLI's SLIs are [127, 112], [111, 96], ... ,[79, 64], totaling 8 intervals. When allocating memory, the corresponding SLI's next SLI will be searched to see if any memory is available; if not, then it will search the bigger SLIs to see if there are any available. When freeing memory, the neighboring available blocks will be combined into a larger block and inserted into the corresponding SLI to prepare for the next allocation.

In RMP's TLSF implementation, the number of FLIs will be decided by the size of the memory pool, while the number of SLIs are always 8. The FLI 0 corresponds to 127-64 Byte blocks, and the memory pool size shall be bigger than 1024 machine words[1]. This memory allocator is optimized for memory pools that are less than 128MB, and it is not recommended to use this for larger pools. Additionally, the memory pool address and size shall be aligned to a machine word, and the smallest allocation size is always 64Byte.

The initialization and usage of memory pools are very flexible[2], thus RMP's memory operations do not lock the scheduler by default. If locking the scheduler during operation is

[1] For 16-bit machines the pool shall not be smaller than 2kB and for 32-bit machines 4kB

[2] The threads can use their own private pools or share them

desired, the feature can be implemented by guarding the operations with RMP_Lock_Sched and RMP_Unlock_Sched.

Last but not least, the memory allocator does not implement protection of any kind, and is unable to recover from any corruption. Therefore, if buffer overflow happens, it may destroy the allocator data structure and make the memory scheme unrecoverable.

The TLSF allocator of RMP merely included 4 function calls, which are listed hereinafter.

2.4.1 Memory Pool Initialization

This operation initializes the memory pool according to the address and size passed in. Both parameters must be aligned to a machine word.

Table 2-20 Memory Pool Initialization

Prototype	rmp_ret_t RMP_Mem_Init(volatile void* Pool, rmp_ptr_t Size)	
Return	rmp_ret_t If successful, 0. If failed, one the following values will be returned:	
	RMP_ERR_MEM	The memory pool is 0 (NULL) or the address/size is not aligned or the size is smaller than 4096 machine words or bigger than 128MB.
Parameter	volatile void* Pool The pointer to the blank memory to initialize as a memory pool.	
	rmp_ptr_t Size The size of the memory pool.	

2.4.2 Memory Allocation

This operation will try to allocate memory from a memory pool. The minimum amount of allocation is 64 Byte. If the Size passed in is smaller than 64 and bigger than 0, the 64 Byte will be allocated; if the Size passed in is 0, the allocation will always fail.

Table 2-21 Memory Allocation

Prototype	void* RMP_Malloc(volatile void* Pool, rmp_ptr_t Size)
Return	void* If successful, the pointer to the memory block. If failed, 0 (NULL).
Parameter	volatile void* Pool The pointer to the memory pool to allocate from.

	rmp_ptr_t Size The memory to allocate, in bytes.

2.4.3 Memory Freeing

This operation will return memory to a memory pool. The pointer passed in must be the exact value returned by a previous RMP_Malloc call, and the memory must be returned to the same pool where it was allocated from.

Table 2-22 Free Memory

Prototype	void RMP_Free(volatile void* Pool, void* Mem_Ptr)
Return	None.
Parameter	volatile void* Pool The pointer to the memory pool to return memory to.
	void* Mem_Ptr The pointer to the memory block to return.

2.4.4 Memory Reallocation

This operation will try to resize the memory block to the Size that is needed. The pointer passed in must be the exact value returned by a previous RMP_Malloc call, and the memory pool must be the same pool where the memory block was allocated from. When successful, this function will return a new memory block whose first Size byte have the same content with the old memory block; else 0 (NULL) will be returned and nothing will happen on the old memory block. Moreover, if Size is 0, this function behaves like RMP_Free and will directly free the memory block; if Mem_Ptr is 0, this function behaves like RMP_Malloc and will directly allocate new memory.

Table 2-23 Memory Reallocation

函数原型	void* RMP_Realloc(volatile void* Pool, void* Mem_Ptr, rmp_ptr_t Size)
Return	void* If successful, return a non-zero (not NULL) pointer to the resized memory block; else return 0 (NULL).
Parameter	volatile void* Pool The pointer to the memory pool which the old memory block belongs to.
	void* Mem_Ptr

	The pointer to the memory block to be resized.
	rmp_ptr_t Size The desired new size of the memory block. Can be smaller or bigger than the original size.

2.5 Other System Interfaces

Other interfaces provided by RMP includes interrupt enabling & disabling, scheduler locking & unlocking and some other helper functions. These functions might be useful in particular application developments.

2.5.1 Interrupt and Scheduler Interfaces

In RMP, the interrupt system interface only included a pair of functions to enable & disable interrupts. This pair of functions is supplied to the user because on some architectures RMP never completely disables interrupts at any point. It is not recommended to use this pair of function unless necessary because it degrades the real-time performance of the system heavily.

The scheduler interface included a pair of functions that can lock & unlock the scheduler with nesting count capabilities. Locking the scheduler also degrades system performance to some degree. Thus it is recommended that the scheduler not be locked where unnecessary. The rule of the thumb is, if scheduler locking is unnecessary, then don't lock the scheduler; if the problem can be solved by locking the scheduler, then don't disable interrupts. Disabling interrupts is a last-ditch approach, and the section should be kept as short as possible.

2.5.1.1 Disabling Interrupts

This operation disables the system's response to all interrupts, including the system tick timer interrupt and the system scheduling interrupt. This function does not have built-in nesting count capabilities, and this functionality should be supplied by the user where necessary.

Table 2-24 Disabling Interrupts

Prototype	void RMP_Disable_Int(void)
Return	None.
Parameter	None.

2.5.1.2 Enabling Interrupts

This operation re-enables the system's response to all interrupts. This function does not have built-in nesting count capabilities, and this functionality should be supplied by the user where necessary.

Table 2-25 Enabling Interrupts

Prototype	void RMP_Enable_Int(void)
Return	None.
Parameter	None.

2.5.1.3 Locking Scheduler

This operation locks the scheduler and thus no other threads can be scheduled. This function have built-in nesting count functionality.

Table 2-26 Locking Scheduler

Prototype	void RMP_Lock_Sched(void)
Return	None.
Parameter	None.

2.5.1.4 Unlocking Scheduler

This operation unlocks the scheduler so that a new thread can be scheduled. This function have built-in nesting count functionality.

Table 2-27 Unlock Scheduler

Prototype	void RMP_Unlock_Sched(void)
Return	None.
Parameter	None.

2.5.2 Helper Functions

To facilitate user application development, RMP provided a useful library, including memory zeroing, debug information printing and linked list operations. The list of these functions are as follows:

2.5.2.1 Clearing Memory

This operation zeros a segment of memory.

Table 2-28 Clearing Memory

Prototype	void RMP_Clear(volatile void* Addr, rmp_ptr_t Size)
Return	None.
Parameter	volatile void* Addr The start address of the memory to zero.
	rmp_ptr_t Size The size of the memory to zero, in bytes.

2.5.2.2 Printing Characters

This operation prints a character to the console[1].

Table 2-29 Printing Characters

Prototype	void RMP_Putchar(char Char)
Return	None.
Parameter	char Char The character to send.

2.5.2.3 Printing Signed Integers

This operation will print a signed integer to the console.

Table 2-30 Printing Signed Integers

Prototype	rmp_cnt_t RMP_Print_Int(rmp_cnt_t Int)
Return	rmp_cnt_t The number of characters printed.
Parameter	rmp_cnt_t Int The integer to print.

2.5.2.4 Printing Unsigned Integers

This operation will print a hexadecimal unsigned integer to the console.

Table 2-31 Printing Unsigned Integers

Prototype	rmp_cnt_t RMP_Print_Uint(rmp_ptr_t Uint)

[1] Usually a serial port interface

Return	rmp_cnt_t The number of characters printed.
Parameter	rmp_ptr_t Uint The unsigned integer to print.

2.5.2.5 Printing Strings

This operation will print a string with a maximum length of 255 bytes to the console.

Table 2-32 Printing Strings

Prototype	rmp_cnt_t RMP_Print_String(rmp_s8_t* String)
Return	rmp_cnt_t The number of characters printed.
Parameter	rmp_s8_t* String The string to print.

2.5.2.6 Getting the Most Significant Bit Position

This operation will get a word's most significant bit's (MSB) position[1]. If the number is zero, -1[2] will be returned.

Table 2-33 Getting the Most Significant Bit Position

Prototype	rmp_ptr_t RMP_MSB_Get(rmp_ptr_t Val)
Return	rmp_ptr_t The MSB's position.
Parameter	rmp_ptr_t Val The unsigned integer to get the MSB.

2.5.2.7 Getting the Least Significant Bit

This operation will get a word's least significant bit's (LSB) position[3]. If the number is zero, then a value equal to the processor word length[4] will be returned.

[1] On 32-bit machines, it should return 0-31

[2] For 32-bit machines, 0xFFFFFFFF

[3] For 32-bit machines, it should return 0-31

[4] For 32-bit machines, 32

Table 2-34 Getting the Least Significant Bit

Prototype	rmp_ptr_t RMP_LSB_Get(rmp_ptr_t Val)
Return	rmp_ptr_t The LSB's position.
Parameter	rmp_ptr_t Val The unsigned integer to get the LSB.

2.5.2.8 Bit Reversal

This operation will reverse the bits of a word. On 32-bit processors this will cause the bit 0 to exchange with bit 31, the bit 1 to exchange with bit 30, the bit 2 to exchange with bit 29, and the same rule applies to all the other bits.

Table 2-35 Bit Reversal

Prototype	rmp_ptr_t RMP_RBIT_Get(rmp_ptr_t Val)
Return	rmp_ptr_t The result after bit reversal.
Parameter	rmp_ptr_t Val The unsigned integer to reverse the bits.

2.5.2.9 List Creation

This operation will initialize the list head of the doubly-linked list.

Table 2-36 List Creation

Prototype	void RMP_List_Crt(volatile struct RMP_List* Head)
Return	无。
Parameter	volatile struct RMP_List* Head The pointer to the list head structure to initialize.

2.5.2.10 List Node Deletion

This operation will delete a node (or a series of nodes) from a doubly linked list.

Table 2-37 List Node Deletion

Prototype	void RMP_List_Del(volatile struct RMP_List* Prev, volatile struct RMP_List* Next)
Return	None.

Parameter	volatile struct RMP_List* Prev The pointer to the previous node of the node(s) to delete.
	volatile struct RMP_List* Next The pointer to the next node of the node(s) to delete.

2.5.2.11 List Node Insertion

This operation will insert a node into a doubly linked list.

Table 2-38 List Node Insertion

Prototype	void RMP_List_Ins(volatile struct RMP_List* New, volatile struct RMP_List* Prev, volatile struct RMP_List* Next)
Return	None.
Parameter	volatile struct RMP_List* New The pointer to the new node to insert into the list.
	volatile struct RMP_List* Prev The pointer to the previous position of the place where the node is to be inserted.
	volatile struct RMP_List* Next The pointer to the next position of the place where the node is to be inserted.

2.5.2.12 CRC16 Calculation

This function will calculate the CRC16 checksum of a memory segment with the polynomial 0xA001. Only when type rmp_u8_t & rmp_u16_t are typedef'ed and macro __RMP_U8_T__ & __RMP_U16_T__ are defined will this function be available.

Table 2-39 CRC16 Calculation

Prototype	rmp_ptr_t RMP_CRC16(const rmp_u8_t* Data, rmp_ptr_t Length)
Return	rmp_ptr_t The resulting CRC16 value.
Parameter	const rmp_u8_t* Data The pointer to the data to calculate CRC16 for.
	rmp_ptr_t Length The length of the data, in bytes.

2.5.3 Hook Function Interfaces

To make it easier to hook user functions into the kernel routines without explicit code modification, RMP provided a series of hook functions to import user code. To use these hooks, the macro RMP_USE_HOOKS needs to be defined as RMP_TRUE. The list of hook functions are listed as follows:

2.5.3.1 System Startup Hook

This hook function will be called immediately after system initialization. Due to the fact that each port will initialize some basic hardware on startup, it is recommended to put further initialization into this hook. If the desired basic system setup conflicts with what is provided in the platform's basic initialization sequence[1], reinitialization of these basic hardware can also be performed here.

Table 2-40 System Startup Hook

Prototype	void RMP_Start_Hook(void)
Return	None.
Parameter	None.

2.5.3.2 Context Save Hook

This hook function will be called after the system have performed its basic context save routine. Should there be more registers to save[2], they can be pushed to stack or be written to some other memory that is associated with the task. Refer to chapter 4 for implementation details.

Table 2-41 Context Save Hook

Prototype	void RMP_Save_Ctx(void)
Return	None.
Parameter	None.

2.5.3.3 Context Load Hook

This hook function will be called before the system performs its basic context load routine. Should there be more registers to load[3], they can be popped from stack or be read from some other memory that is associated with the task. Refer to chapter 4 for implementation details.

[1] E.g. configuring the system for different operating frequencies

[2] Peripheral state or FPU registers

[3] Peripheral state or FPU registers

Table 2-42 Context Load Hook

Prototype	void RMP_Load_Ctx(void)
Return	None.
Parameter	None.

2.5.3.4 Scheduler Hook

This hook function will be called whenever there is a scheduler invoke. This can be used to implement a tickless system. Refer to 6.1.1 for implementation details.

Table 2-43 Scheduler Hook

Prototype	void RMP_Load_Ctx(void)
Return	None.
Parameter	None.

2.5.3.5 Tick Timer Hook

This hook function will be called whenever there is a system tick. This can be used to implement a tickless system. Refer to 6.1.1 for implementation details.

Table 2-44 Tick Timer Hook

Prototype	void RMP_Tick_Hook(rmp_ptr_t Ticks)
Return	None.
Parameter	rmp_ptr_t Ticks The number of ticks passed from the last tick interrupt.

2.5.3.6 Init Thread Initialization Hook

This hook function will be called once when the initial thread runs for the first time, and by default the scheduler is locked in this function. This hook will always be enabled regardless of the RMP_USE_HOOKS macro. It is recommended to use this function to create threads and semaphores.

Table 2-45 Init Thread Initialization Hook

Prototype	void RMP_Init_Hook(void)
Return	None.
Parameter	None.

2.5.3.7 Init Thread Loop Hook

This hook function will be called repeatedly when the initial thread runs. This hook will always be enabled regardless of the RMP_USE_HOOKS macro. It is recommended to use this function to put the processor into sleep state, or do performance auditing and stack monitoring.

Table 2-46 Init Thread Loop Hook

Prototype	void RMP_Init_Idle(void)
Return	None.
Parameter	None.

2.6 Bibliography

[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.

Chapter 3 Light-Weight Graphics Library

3.1 Introduction to Embedded Graphic User Interfaces

Embedded Graphic User Interfaces (GUIs) are GUIs that are designed specifically for embedded systems. Compared to traditional GUI systems, the embedded GUI systems have four distinct idiosyncrasies: simplicity, diversity, planarity and restrictiveness. Keeping the four idiosyncrasies in mind is the key to successful embedded GUI designs.

3.1.1 Simplicity

In most cases, traditional GUIs have scroll bars and tree views, which are designed for complex user interfaces. On the contrary, the composing elements of embedded GUI are relatively simple, which are often a few buttons and display widgets. Moreover, once the embedded GUI design phase is complete, the interface itself seldom changes.

3.1.2 Diversity

After many years of evolution, traditional GUIs have converged to a common style, which is seen in many common desktop environments such as Windows, MacOS, Gnome and Unity. All of them are optimized for keyboard and mouse operation experiences, and have standardized widgets. However, in embedded GUIs it is just the contrary. Mouses and keyboards are rarely used, while touch screens and customized keypads are prevalent. More customized widgets are required for different purposes, and they need to be optimized or customized for their respective applications.

3.1.3 Planarity

Traditional GUIs emphasize vertical sequence of multiple windows. To calculate the display area of the windows, sophisticated window clipping algorithms must be applied. This is necessary in multi-window multi-task environments; however in embedded GUI realm this is not the case. For embedded GUIs, the display area is usually very small, and means of input are usually very limited. This renders complex vertical sequence of windows useless, and more effort is put into optimizing for a planar design. This trend can also be discovered by investigating the evolution of smart phone GUIs.

3.1.4 Restrictiveness

Traditional GUIs perform complicated graphics operations which are backed by powerful CPUs and GPUs. In embedded GUI environments however, only a mediocre CPU is available, let

alone dedicated graphics hardware. Sometimes even making space for frame buffers is difficult. When designing embedded GUIs, this factor also needs to be taken into consideration.

3.2 Embedded GUI Support of RMP

RMP supports various GUI functionality by providing a series of graphics rendering functions. The basics includes dot and line rendering, polygon rendering and bitmap anti-aliasing rendering; more advanced features include many widgets. All of these interfaces do not store any internal information about the GUI object, and are not designed in a object-oriented manner; thus, when the GUI feature is enabled, only ROM is consumed. To use the GUI components, a few extra macros must be defined in the configuration header, and only when they are defined will the corresponding function be available. A comprehensive list of these macros are as follows:

Table 3-1 Macros Required when Using Built-in GUI Support

Name	Explanation
RMP_POINT	The macro for drawing a point. This macro is used across all GUI functions, and must be defined to use the GUI module. This macro shall be defined as the point drawing function's name, and the function must have the following prototype: void foo(rmp_cnt_t X, rmp_cnt_t Y, rmp_ptr_t Color); This function shall draw a point on the screen. Example: #define RMP_POINT RMP_Point
RMP_COLOR_25P(C1, C2) RMP_COLOR_50P(C1, C2) RMP_COLOR_75P(C1, C2)	The macro for mixing colors. Defining these macros are only required for drawing anti-aliased bitmaps. All three macros carry two colors as their argument, and shall return a color. RMP_COLOR_25P means mixing C1 and C2 with a 25%+75% ratio, RMP_COLOR_50P means mixing C1 and C2 in half, RMP_COLOR_75Pmeans mixing C1 and C2 with a 75%+25% ratio.
RMP_CTL_WHITE RMP_CTL_LGREY RMP_CTL_GREY RMP_CTL_DGREY RMP_CTL_DARK RMP_CTL_DDARK RMP_CTL_BLACK	The macro for built-in widget colors. Defining them is only necessary when using built-in widgets. RMP_CTL_WHITE white, #FFFFFF recommended. RMP_CTL_LGREY light grey, #B8B8B8 recommended. RMP_CTL_GREY grey, #E0E0E0 recommended. RMP_CTL_DGREY dark grey, #A0A0A0 recommended. RMP_CTL_DARK dark, #787C78 recommended. RMP_CTL_DDARK darker dark, #686868 recommended.

Name	Explanation
	RMP_CTL_BLACK black, #000000 recommended.

The functions provided by RMP will not check whether the X and Y coordinates of the widgets are out of the border. It is the developer's duty to check this and take actions when needed. The comprehensive list of functions is shown hereinafter:

3.2.1 Drawing Lines

This function will draw a line on the screen with Bresenham algorithm. The width of the line is always 1 pixel.

Table 3-2 Drawing Lines

Prototype	void RMP_Line(rmp_cnt_t Start_X, rmp_cnt_t Start_Y, rmp_cnt_t End_X, rmp_cnt_t End_Y, rmp_ptr_t Color)
Return	None.
Parameter	rmp_cnt_t Start_X The X coordinate of the starting point of the line.
	rmp_cnt_t Start_Y The Y coordinate of the starting point of the line.
	rmp_cnt_t End_X The X coordinate of the ending point of the line.
	rmp_cnt_t End_Y The Y coordinate of the ending point of the line.
	rmp_ptr_t Color The color of the line.

3.2.2 Drawing Dotted Lines

This function will draw a dotted line on the screen with Bresenham algorithm. The width of the dotted line is always 1 pixel.

Table 3-3 Drawing Dotted Lines

Prototype	void RMP_Dot_Line(rmp_cnt_t Start_X, rmp_cnt_t Start_Y, rmp_cnt_t End_X, rmp_cnt_t End_Y, rmp_ptr_t Dot, rmp_ptr_t Space)
Return	None.

Parameter	rmp_cnt_t Start_X The X coordinate of the starting point of the dotted line.
	rmp_cnt_t Start_Y The Y coordinate of the starting point of the dotted line.
	rmp_cnt_t End_X The X coordinate of the ending point of the dotted line.
	rmp_cnt_t End_Y The Y coordinate of the ending point of the dotted line.
	rmp_ptr_t Dot The color of the dots.
	rmp_ptr_t Space The color of the spaces. If RMP_TRANS, the spaces will not be drawn.

3.2.3 Drawing Rectangles

This function will draw a rectangle on the screen. The rectangle can have a different border color than the filling color, and the filling is optional.

Table 3-4 Drawing Rectangles

Prototype	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)
Return	None.
Parameter	rmp_cnt_t Coord_X The X coordinate of the top-left corner of the rectangle.
	rmp_cnt_t Coord_Y The Y coordinate of the top-left corner of the rectangle.
	rmp_cnt_t Length The length of the rectangle.
	rmp_cnt_t Width The width of the rectangle.
	rmp_ptr_t Border The border color of the rectangle.
	rmp_ptr_t Fill The color of the filling. If RMP_TRANS, the rectangle will not be filled.

3.2.4 Drawing Rounded Rectangles

This function will draw a rounded rectangle on the screen. The border color is always the same with the filling color, and the rounded rectangle must be filled.

Table 3-5 Drawing Rounded Rectangles

Prototype	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)
Return	None.
Parameter	rmp_cnt_t Coord_X The X coordinate of the top-left corner of the rounded rectangle.
	rmp_cnt_t Coord_Y The Y coordinate of the top-left corner of the rounded rectangle.
	rmp_cnt_t Length The length of the rounded rectangle.
	rmp_cnt_t Width The width of the rounded rectangle.
	rmp_cnt_t Round The radius of the rounded corner.
	rmp_ptr_t Color The color of the rounded rectangle.

3.2.5 Drawing Circles

This function will draw a circle on the screen. The circle border can have a different color than the filling color, and the filling is optional.

Table 3-6 Drawing Circles

Prototype	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)
Return	None.
Parameter	rmp_cnt_t Center_X The X coordinate of the center of the circle.
	rmp_cnt_t Center_Y The Y coordinate of the center of the circle.
	rmp_cnt_t Radius The radius of the circle.

	rmp_ptr_t Border The border color of the circle.
	rmp_ptr_t Fill The color of the filling. If RMP_TRANS, the circle will not be filled.

3.2.6 Drawing Monochrome Bitmaps

This function will draw a monochrome bitmap to the screen. The length of the bitmap must be a multiple of 8, and must be stored in a left-to-right, up-to-down manner. When rendering the bitmap, the bit order can be configured as high-order or low-order. When the former is selected, the MSB in a byte represents the pixel with lower X value, and the latter is just the opposite[1].

Table 3-7 Drawing Monochrome Bitmaps

Prototype	void RMP_Matrix(rmp_cnt_t Coord_X, rmp_cnt_t Coord_Y, const rmp_u8_t* Matrix, rmp_cnt_t Bit_Order, rmp_cnt_t Length, rmp_cnt_t Width, rmp_ptr_t Color)
Return	None.
Parameter	rmp_cnt_t Coord_X The X coordinate of the top-left corner of the bitmap.
	rmp_cnt_t Coord_Y The Y coordinate of the top-left corner of the bitmap.
	const rmp_u8_t* Matrix The pointer to the monochrome bitmap data.
	rmp_cnt_t Bit_Order The bit order in a byte. RMP_MAT_BIG for high-order, 0 for low-order.
	rmp_cnt_t Length The length of the bitmap, must be a multiple of 8 or the bitmap will not be rendered.
	rmp_cnt_t Width The width of the bitmap.
	rmp_ptr_t Color The color to use for this bitmap.

3.2.7 Drawing Monochrome Bitmaps with Anti-Aliasing

[1] This concept is very similar to endianness; but this is referring to the pixel sequence in a byte

This function will draw a monochrome bitmap with 4xFXAA to the screen. The length of the bitmap must be a multiple of 8, and must be stored in a left-to-right, up-to-down manner. When rendering the bitmap, the bit order can be configured as high-order or low-order. The background color also needs to be passed in for anti-aliasing. Additionally, all three color mixing macros must be defined to use this function.

Table 3-8 Drawing Monochrome Bitmaps with Anti-Aliasing

Prototype	void RMP_Matrix_AA(rmp_cnt_t Coord_X, rmp_cnt_t Coord_Y, const rmp_u8_t* Matrix, rmp_cnt_t Bit_Order, rmp_cnt_t Length, rmp_cnt_t Width, rmp_ptr_t Color, rmp_ptr_t Back)
Return	None.
Parameter	rmp_cnt_t Coord_X The X coordinate of the top-left corner of the bitmap.
	rmp_cnt_t Coord_Y The Y coordinate of the top-left corner of the bitmap.
	const rmp_u8_t* Matrix The pointer to the monochrome bitmap data.
	rmp_cnt_t Bit_Order The bit order in a byte. RMP_MAT_BIG for high-order, 0 for low-order.
	rmp_cnt_t Length The length of the bitmap, must be a multiple of 8 or the bitmap will not be rendered.
	rmp_cnt_t Width The width of the bitmap.
	rmp_ptr_t Color The color to use for this bitmap.
	rmp_ptr_t Back The background canvas' color.

3.2.8 Drawing Cursors

This function will draw a 16x16 cursor on the screen. To use this function and all the functions that follow this function, all widget color macros must be defined.

Table 3-9 Drawing Cursors

Prototype	void RMP_Cursor(rmp_cnt_t Coord_X, rmp_cnt_t Coord_Y, rmp_ptr_t Style)	
Return	None.	
Parameter	rmp_cnt_t Coord_X	
	The X coordinate of the top-left corner of the cursor.	
	rmp_cnt_t Coord_Y	
	The Y coordinate of the top-left corner of the cursor.	
	rmp_ptr_t Style	
	The style of the cursor. One of the following can be appointed:	
	RMP_CUR_NORM	The normal cursor.
	RMP_CUR_BUSY	The cursor with a busy mark.
	RMP_CUR_QUESTION	The cursor with a question mark.
	RMP_CUR_HAND	The hand-shaped cursor.
	RMP_CUR_TEXT	The text-editing cursor.
	RMP_CUR_STOP	The “stop” cursor.
	RMP_CUR_MOVE	The “move” cursor.
	RMP_CUR_LR	The left-to-right adjustment cursor.
	RMP_CUR_UD	The up-to-down adjustment cursor.
	RMP_CUR_ULBR	The bottom-right to upper-left adjustment cursor.
	RMP_CUR_URBL	The bottom-left to upper-right adjustment cursor.
	RMP_CUR_CROSS	The cross mark cursor.

3.2.9 Drawing Checkboxs

This function will draw a scalable checkbox on the screen.

Table 3-10 Drawing Checkboxs

Prototype	void RMP_Checkbox(rmp_cnt_t Coord_X, rmp_cnt_t Coord_Y, rmp_cnt_t Length, rmp_ptr_t Status)
Return	None.
Parameter	rmp_cnt_t Coord_X
	The X coordinate of top-left corner of the checkbox.
	rmp_cnt_t Coord_Y

	The Y coordinate of top-left corner of the checkbox.
	rmp_cnt_t Length The length[1] of the checkbox.
	rmp_ptr_t Status The status of the checkbox. RMP_CBOX_CHECK for checked, 0 for unchecked.

3.2.10 Setting Checkboxes

This function will check a checkbox that is already drawn.

Table 3-11 Setting Checkboxes

Prototype	void RMP_Checkbox_Set(rmp_cnt_t Coord_X, rmp_cnt_t Coord_Y, rmp_cnt_t Length)
Return	None.
Parameter	rmp_cnt_t Coord_X The X coordinate of top-left corner of the checkbox.
	rmp_cnt_t Coord_Y The Y coordinate of top-left corner of the checkbox.
	rmp_cnt_t Length The length (width) of the checkbox.

3.2.11 Clearing Checkboxes

This function will clear a checkbox that is already drawn.

Table 3-12 Clearing Checkboxes

Prototype	void RMP_Checkbox_Clr(rmp_cnt_t Coord_X, rmp_cnt_t Coord_Y, rmp_cnt_t Length)
Return	None.
Parameter	rmp_cnt_t Coord_X The X coordinate of top-left corner of the checkbox.
	rmp_cnt_t Coord_Y The Y coordinate of top-left corner of the checkbox.
	rmp_cnt_t Length The length[2] of the checkbox.

[1] Also width

[2] Also width

3.2.12 Drawing Command Buttons

This function will draw a scalable command button.

Table 3-13 Drawing Command Buttons

Prototype	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)
Return	None.
Parameter	rmp_cnt_t Coord_X The X coordinate of top-left corner of the command button.
	rmp_cnt_t Coord_Y The Y coordinate of top-left corner of the command button.
	rmp_cnt_t Length The length of the command button.
	rmp_cnt_t Width The width of the command button.
	rmp_ptr_t Status The status of the command button. RMP_CBTN_DOWN for pushed, 0 for not pushed.

3.2.13 Pushing Command Buttons

This function will push a drawn command button.

Table 3-14 Pushing Command Buttons

Prototype	void RMP_Cmdbtn_Down(rmp_cnt_t Coord_X, rmp_cnt_t Coord_Y, rmp_cnt_t Length, rmp_cnt_t Width)
Return	None.
Parameter	rmp_cnt_t Coord_X The X coordinate of top-left corner of the command button.
	rmp_cnt_t Coord_Y The Y coordinate of top-left corner of the command button.
	rmp_cnt_t Length The length of the command button.
	rmp_cnt_t Width The width of the command button.

3.2.14 Releasing Command Buttons

This function will pop up a drawn command button.

Table 3-15 Releasing Command Buttons

Prototype	void RMP_Cmdbtn_Up(rmp_cnt_t Coord_X, rmp_cnt_t Coord_Y, rmp_cnt_t Length, rmp_cnt_t Width)
Return	None.
Parameter	rmp_cnt_t Coord_X The X coordinate of top-left corner of the command button.
	rmp_cnt_t Coord_Y The Y coordinate of top-left corner of the command button.
	rmp_cnt_t Length The length of the command button.
	rmp_cnt_t Width The width of the command button.

3.2.15 Drawing Text Editing Boxes

This function will draw a scalable text edit box.

Table 3-16 Drawing Text Editing Boxes

Prototype	void RMP_Lineedit(rmp_cnt_t Coord_X,rmp_cnt_t Coord_Y, rmp_cnt_t Length, rmp_cnt_t Width)
Return	None.
Parameter	rmp_cnt_t Coord_X The X coordinate of top-left corner of the text edit box.
	rmp_cnt_t Coord_Y The Y coordinate of top-left corner of the text edit box.
	rmp_cnt_t Length The length of the text edit box.
	rmp_cnt_t Width The width of the text edit box.

3.2.16 Clearing a Portion of Text Editing Boxes

This function will clear a portion of the text edit box.

Table 3-17 Clearing a Portion of Text Editing Boxes

Prototype	void RMP_Lineedit_Clr(rmp_cnt_t Coord_X, rmp_cnt_t Coord_Y, rmp_cnt_t Length, rmp_cnt_t Width, rmp_cnt_t Clr_X, rmp_cnt_t Clr_Len)
Return	None.
Parameter	rmp_cnt_t Coord_X The X coordinate of top-left corner of the text edit box.
	rmp_cnt_t Coord_Y The Y coordinate of top-left corner of the text edit box.
	rmp_cnt_t Length The length of the text edit box.
	rmp_cnt_t Width The width of the text edit box.
	rmp_cnt_t Clr_X The relative X coordinate of the position to start clearing.
	rmp_cnt_t Clr_Len The length to clear.

3.2.17 Drawing Radio Buttons

This function will draw a scalable radio button.

Table 3-18 Drawing Radio Buttons

Prototype	void RMP_Radiobtn(rmp_cnt_t Coord_X, rmp_cnt_t Coord_Y, rmp_cnt_t Length, rmp_ptr_t Status)
Return	None.
Parameter	rmp_cnt_t Coord_X The X coordinate of top-left corner of the radio button.
	rmp_cnt_t Coord_Y The Y coordinate of top-left corner of the radio button.
	rmp_cnt_t Length The length[1] of the radio button.
	rmp_ptr_t Status

[1] Also width

	The status of the radio button. RMP_RBTN_SEL for selected, 0 for not selected.

3.2.18 Setting Radio Buttons

This function will select a drawn radio button.

Table 3-19 Setting Radio Buttons

Prototype	void RMP_Radiobtn_Set(rmp_cnt_t Coord_X, rmp_cnt_t Coord_Y, rmp_cnt_t Length)
Return	None.
Parameter	rmp_cnt_t Coord_X The X coordinate of top-left corner of the radio button.
	rmp_cnt_t Coord_Y The Y coordinate of top-left corner of the radio button.
	rmp_cnt_t Length The length[1] of the radio button.

3.2.19 Clearing Radio Buttons

This function will clear a drawn radio button.

Table 3-20 Clearing Radio Buttons

Prototype	void RMP_Radiobtn_Clr(rmp_cnt_t Coord_X, rmp_cnt_t Coord_Y, rmp_cnt_t Length)
Return	None.
Parameter	rmp_cnt_t Coord_X The X coordinate of top-left corner of the radio button.
	rmp_cnt_t Coord_Y The Y coordinate of top-left corner of the radio button.
	rmp_cnt_t Length The length (width) of the radio button.

3.2.20 Drawing Progress Bars

This function will draw a progress bar on the screen.

Table 3-21 Drawing Progress Bars

[1] Also width

Prototype	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)
Return	None.
Parameter	rmp_cnt_t Coord_X The X coordinate of top-left corner of the progress bar.
	rmp_cnt_t Coord_Y The Y coordinate of top-left corner of the progress bar.
	rmp_cnt_t Length The length of the progress bar.
	rmp_cnt_t Width The width of the progress bar.
	rmp_cnt_t Style The style of the progress bar. Can be one of the following:
	RMP_PBAR_L2R — Grows from left to right.
	RMP_PBAR_D2U — Grows from down to up.
	RMP_PBAR_R2L — Grows from right to left.
	RMP_PBAR_U2D — Grows from up to down.
	rmp_cnt_t Prog The progress of the progress bar, must be between 0 and 100.
	rmp_ptr_t Fore The foreground of the progress bar, that is, the color of the progress.
	rmp_ptr_t Back The background of the progress bar, that is, the color of the empty region.

3.2.21 Setting Progress of Progress Bars

This function will change the progress of a drawn progress bar.

Table 3-22 Setting Progress of Progress Bars

Prototype	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)
Return	None.

Parameter	rmp_cnt_t Coord_X The X coordinate of top-left corner of the progress bar.	
	rmp_cnt_t Coord_Y The Y coordinate of top-left corner of the progress bar.	
	rmp_cnt_t Length The length of the progress bar.	
	rmp_cnt_t Width The width of the progress bar.	
	rmp_cnt_t Style The style of the progress bar. Can be one of the following:	
	RMP_PBAR_L2R	Grows from left to right.
	RMP_PBAR_D2U	Grows from down to up.
	RMP_PBAR_R2L	Grows from right to left.
	RMP_PBAR_U2D	Grows from up to down.
	rmp_cnt_t Old_Prog The old progress of the progress bar, must be between 0 and 100.	
	rmp_cnt_t New_Prog The new progress of the progress bar, must be between 0 and 100.	
	rmp_ptr_t Fore The foreground of the progress bar (the color of the progress).	
	rmp_ptr_t Back The background of the progress bar (the color of the empty region).	

3.3 GUI Design Guidelines

Some guidelines needs to be followed when designing GUIs due to the restrictions of the library. Some tricks are also needed to ensure a good design outcome. These guidelines will be explained in detail in the following sections.

3.3.1 Planar Design

Due to the fact that window manager is not supported by this library, a planar design style is recommended. For detailed concepts of this style, please refer to internet resources and related articles, such as the UI design guides of the Apple Inc.

3.3.2 Use Simple Background and Bitmaps

Due to the fact that this GUI does not support display layers and alpha blending, its anti-aliasing bitmap rendering function requires a background color as a input. If a non-unicolor background is used, this function cannot be used. In most cases complex background is not required, and what is needed is simply a dark grey background.

If possible, avoid full-color pictures as well. This is mainly due to storage constraints on Flash ROM. Using monochrome bitmaps can save up to 90% storage when compared to colored pictures.

3.3.3 Design and Implementation of Character Libraries

There are two ways of implementing the character libraries when using this GUI library. One is to pick only the characters used and render them as monochrome bitmaps, which is stored in the ROM of the microcontroller. The other one is to use external dedicated character library chips. For applications that emphasize aesthetic values and do not require a full character set, the former is recommended; for applications that require display of a full character set, the latter is recommended.

3.3.4 Design and Implementation of Complex Widgets

This GUI only implemented basic widgets and is unable to meet very high aesthetic standards. When this is the case, customized widgets must be designed. To implement custom widgets, the bitmap rendering functions can be used and the bitmap of the widgets should be updated by the user upon detection of state changes.

3.4 Bibliography

None

Chapter 4 Formal Verification

4.1 Introduction to Formal Verification

The definitive idiosyncrasies of an embedded real-time operating systems are complexity, diversity and reliability. The hardware-independent part of the RMP operating system includes approximately 2000 lines of code, with complex functions and connections between the various code sections. As an operating system, RMP needs to meet the various needs of different applications yet still must guarantee the correctness of the functionality. Meanwhile, we also need to improve development efficiency, control development costs and development cycles, and ensure on-time delivery of software.

Traditional software design techniques are based on natural language thinking, designing and description. They are often unscientific and vague, and will easily cause misunderstanding. Such development processes can only be analyzed by human developers, and cannot be automatically examined. Based on a relatively clear, graphical depiction of software system, some semi-formal methods such as UML can also automatically generate the code framework and examine the analysis. The two methods mentioned can only support use case testing, and in some cases they are automatic or structured. However, none of them can guarantee that there are no errors in the system and are not suitable for the analysis and development of safety-critical systems. Therefore, we need to apply a more rigorous design approach. Formal methods are based on well-defined mathematical concepts and definitions and can leverage fully automated tools for inspection and analysis. It brings the rigorousness of mathematics into all phases of software development, and only with rigorous mathematical proofs, we can say that that there are no vulnerabilities in the system.

The EAL standard of software safety have 7 levels, listed hereinafter[1][1]:

Table 4-1 EAL Levels and Descriptions

EAL Lvl.	Description
EAL1	EAL1 provides a basic level of assurance by a limited security target and an analysis of the SFRs (Specification of Functional Request) in that ST (Security Target) using a functional and interface specification and guidance documentation, to understand the security behaviour. The analysis is supported by a search for potential vulnerabilities in the public domain and independent testing (functional and penetration) of the TSF (TOE Security Function).

[1] EAL7+ is also inluded as a separate level

	EAL1 also provides assurance through unique identification of the TOE (Target of Evaluation) and of the relevant evaluation documents. This EAL provides a meaningful increase in assurance over unevaluated IT.
EAL2	EAL2 provides assurance by a full security target and an analysis of the SFRs in that ST, using a functional and interface specification, guidance documentation and a basic description of the architecture of the TOE, to understand the security behaviour. The analysis is supported by independent testing of the TSF, evidence of developer testing based on the functional specification, selective independent confirmation of the developer test results, and a vulnerability analysis (based upon the functional specification, TOE design, security architecture description and guidance evidence provided) demonstrating resistance to penetration attackers with a basic attack potential. EAL2 also provides assurance through use of a configuration management system and evidence of secure delivery procedures. This EAL represents a meaningful increase in assurance from EAL1 by requiring developer testing, a vulnerability analysis (in addition to the search of the public domain), and independent testing based upon more detailed TOE specifications.
EAL3	EAL3 provides assurance by a full security target and an analysis of the SFRs in that ST, using a functional and interface specification, guidance documentation, and an architectural description of the design of the TOE, to understand the security behaviour. The analysis is supported by independent testing of the TSF, evidence of developer testing based on the functional specification and TOE design, selective independent confirmation of the developer test results, and a vulnerability analysis (based upon the functional specification, TOE design, security architecture description and guidance evidence provided) demonstrating resistance to penetration attackers with a basic attack potential. EAL3 also provides assurance through the use of development environment controls, TOE configuration management, and evidence of secure delivery procedures. This EAL represents a meaningful increase in assurance from EAL2 by requiring more complete testing coverage of the security functionality and mechanisms and/or procedures that provide some confidence that the TOE will not be tampered with during development.
EAL4	EAL4 provides assurance by a full security target and an analysis of the SFRs in that ST, using a functional and complete interface specification, guidance

	documentation, a description of the basic modular design of the TOE, and a subset of the implementation, to understand the security behaviour. The analysis is supported by independent testing of the TSF, evidence of developer testing based on the functional specification and TOE design, selective independent confirmation of the developer test results, and a vulnerability analysis (based upon the functional specification, TOE design, implementation representation, security architecture description and guidance evidence provided) demonstrating resistance to penetration attackers with an Enhanced-Basic attack potential. EAL4 also provides assurance through the use of development environment controls and additional TOE configuration management including automation, and evidence of secure delivery procedures. This EAL represents a meaningful increase in assurance from EAL3 by requiring more design description, the implementation representation for the entire TSF, and improved mechanisms and/or procedures that provide confidence that the TOE will not be tampered with during development.
EAL5	EAL5 provides assurance by a full security target and an analysis of the SFRs in that ST, using a functional and complete interface specification, guidance documentation, a description of the design of the TOE, and the implementation, to understand the security behaviour. A modular TSF design is also required. The analysis is supported by independent testing of the TSF, evidence of developer testing based on the functional specification, TOE design, selective independent confirmation of the developer test results, and an independent vulnerability analysis demonstrating resistance to penetration attackers with a moderate attack potential. EAL5 also provides assurance through the use of a development environment controls, and comprehensive TOE configuration management including automation, and evidence of secure delivery procedures. This EAL represents a meaningful increase in assurance from EAL4 by requiring semiformal design descriptions, a more structured (and hence analyzable) architecture, and improved mechanisms and/or procedures that provide confidence that the TOE will not be tampered with during development.
EAL6	EAL6 provides assurance by a full security target and an analysis of the SFRs in that ST, using a functional and complete interface specification, guidance documentation, the design of the TOE, and the implementation to understand the security behaviour. Assurance is additionally gained through a formal model of select TOE security policies and a semiformal presentation of the functional specification and TOE design. A modular, layered and simple TSF design is also required.

	The analysis is supported by independent testing of the TSF, evidence of developer testing based on the functional specification, TOE design, selective independent confirmation of the developer test results, and an independent vulnerability analysis demonstrating resistance to penetration attackers with a high attack potential. EAL6 also provides assurance through the use of a structured development process, development environment controls, and comprehensive TOE configuration management including complete automation, and evidence of secure delivery procedures. This EAL represents a meaningful increase in assurance from EAL5 by requiring more comprehensive analysis, a structured representation of the implementation, more architectural structure (e.g. layering), more comprehensive independent vulnerability analysis, and improved configuration management and development environment controls.
EAL7	EAL7 provides assurance by a full security target and an analysis of the SFRs in that ST, using a functional and complete interface specification, guidance documentation, the design of the TOE, and a structured presentation of the implementation to understand the security behaviour. Assurance is additionally gained through a formal model of select TOE security policies and a semiformal presentation of the functional specification and TOE design. A modular, layered and simple TSF design is also required. The analysis is supported by independent testing of the TSF, evidence of developer testing based on the functional specification, TOE design and implementation representation, complete independent confirmation of the developer test results, and an independent vulnerability analysis demonstrating resistance to penetration attackers with a high attack potential. EAL7 also provides assurance through the use of a structured development process, development environment controls, and comprehensive TOE configuration management including complete automation, and evidence of secure delivery procedures. This EAL represents a meaningful increase in assurance from EAL6 by requiring more comprehensive analysis using formal representations and formal correspondence, and comprehensive testing.
EAL7+	EAL7+ provides assurance by a full security target and an analysis of the SFRs in that ST, using a functional and complete interface specification, guidance documentation, the design of the TOE, and a structured presentation of the implementation to understand the security behaviour.

Assurance is additionally gained through a formal model of select TOE security policies and a completely formal presentation of the functional specification and TOE design. A modular, layered and simple TSF design is also required.

The analysis is supported by independent testing of the TSF, evidence of developer testing based on the functional specification, TOE design and implementation representation, complete independent confirmation of the developer test results, and an independent vulnerability analysis demonstrating resistance to penetration attackers with a high attack potential.

EAL7 also provides assurance through the use of a structured development process, development environment controls, and comprehensive TOE configuration management including complete automation, and evidence of secure delivery procedures.

This EAL represents a meaningful increase in assurance from EAL7 by requiring complete analysis using formal representations and formal correspondence, and comprehensive testing.

RMP is designed to meet the EAL7+ standard.

Most operating systems e.g. FreeRTOS, Windows and Linux are certified to the equivalent of EAL4. Some systems are certified to EAL5, and INTEGRITY-178B is certified to EAL6+. Due to the relative simple construction of RMP and the fact that RMP does not need to guarantee security[1], it will be easy to certify it to a very high level.

4.2 Formal Model of the System

In the RMP system, the architecture-dependent part of the system is usually very short[2], and it is easy to guarantee their correctness, thus they are not verified. Another reason is that the architecture dependent part usually changes with the chip design, while RMP aims to support a myriad of architectures. It is very time-consuming and simply impossible to establish formal models for every architecture and verify them one by one. Thus, the bulk of the verification work concentrates on the 1500 Lines Of Code (LOC) in rmp_kernel.c file, of which about 1000 are Source Lines Of Code (SLOC). Due to the same reason, we only verify the correctness of this software to the C programming language semantic level and will not perform any binary verification. This means that the correctness of the final binary is still dependent on the correctness of the compilers. Practically speaking, these compilers are usually certified to a high

[1] Most CC/CAPP/LSPP standards are not applicable

[2] Usually within 50 lines of code

level of confidence, and it is appropriate to trust them. If you do not wish to trust the compiler, formally verified compilers such as CompCert[2] are also available.

Formal verification does not rule out all errors in a cyber-physical system, especially these hardware-introduced ones[1]. The formal verification of RMP does not help in case of a hardware failure or a faulty Hardware Abstraction Layer (HAL) implementation.

4.3 Formal Proof

Future work in progress.

4.4 Other Documents

Future work in progress.

4.5 Bibliography

[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] E.g. CPU or peripheral silicon bugs, which are often found in chip errata sheets from their manufacturers

Chapter 5 Porting RMP to New Architectures

5.1 Introduction to Porting

Porting of an OS refers to the work to modify it to run on new architecture and platform. Sometimes, we call the work to make it possible to use a new compiler porting as well. Compared to the porting process of uC/OS and RT-Thread, the porting of RMP is very simple. All code of RMP is written in MISRA-C compliant ANSI/ISO C89, and contains minimal assembly code. Thus, the porting work includes just a few steps.

Before porting, we need some preparation to make sure that porting is possible. Then we modify the code sections accordingly to make it work on new platforms. After porting, we can run the test bench to validate that porting is successful. Because the architecture-independent portion of RMP is formally verified, any modification to these code is prohibited otherwise the formal verification guarantees will void.

5.2 Checklist Before Porting

5.2.1 Processor

RMP requires that the processor family selected have at last 2 interrupt sources, one of them needs to be connected to a timer, the other one must be software-triggerable. No other requirements are imposed. It makes little sense to run RMP on 8-bit platforms, especially these MCUs with less than 8kB Flash; in these cases RMS state machine OS might be a better choice. If the target platform supports an MMU or have multiple cores, consider running RME microkernel on it. RMP does not support hardware stack architectures; the stack must be implemented in software[1] and not in hardware[2].

5.2.2 Compiler

RMP requires that the compiler is C89 compliant, and can produce code according to some function calling convention. Because the RMP is written with strict accordance to the standard, and did not use any C runtime library components, so it only requires that the compiler is C89 compliant. Common compilers such as GCC, CLANG/LLVM, MSVC, ARMCC, ICC, IAR, TASKING satisfy the requirement very well. Also, RMP does not use compiler extensions that may vary across compiler implementations, such as bitfield, enum and struct packing, and undefined behaviors are avoided as much as possible, thus maximum compatibility can be expected.

[1] Such that the stack pointer and stack content can be software manipulated

[2] E.g. some PIC microcontrollers

When using the compiler, although in most cases maximum optimization will not cause problems, it is recommended to turn off dead code elimination and link-time optimizations, as well as the loop invariant code motion, especially on low-quality compilers. Do not use any aggressive optimization feature. On common compilers, the recommended optimization level is GCC -O2 or its equivalent optimization level.

5.2.3 Assembler

RMP requires that the assembler is capable of exporting symbols to C and importing symbols from C. The assembler must be able to generate code that conform to the calling convention. Usually these are easily satisfied. If the compiler supports inline assembly, then an standalone assembler is not required.

5.2.4 Debugger

RMP does not have any special requirements regarding the debugger . It is best if debuggers are available, however, porting is possible without one. Variables can be inspected with a debugger where available; when not available, it is recommended to implement the low-level RMP_Putchar function first to print characters, then it will be possible to print log with this facility. Please refer to the sections that follows for detailed description about the function.

5.3 Architecture-depedent Portions of RMP

RMP's architecture dependent part is located in the Platform folder's corresponding subfolders. For instance, the Cortex-M architecture's folder is located at Platform/CortexM. The corresponding headers are located at Include/Platform/CortexM, and the same rule applies to all other architectures.

Each architecture includes one or more source files and headers. When the kernel includes a certain architecture's header, it does so by including the file Include/rmp_platform.h, which in turn includes all other headers. When changing RMP's target platform, this file is changed to include corresponding platform's header. For instance, if we are compiling for the Cortex-M architecture, then the header should include the corresponding header(s) of Cortex-M.

Cortex-M port can be used as a template and start the porting work can be started from there.

5.3.1 Typedefs

For each architecture and compiler, the typedefs are always among the first things to be ported. It is worth noting that for some architectures and compilers, the long type corresponds two machine words instead of one; in this case, the int type should be used as a base to

represent a machine word; for some other ones, the int type corresponds to half a machine word's length, it which case long should be used as a base to represent a machine word. For the exact length of every primitive data type that the compiler provided, be sure to write some small programs with sizeof() to confirm their exact length.

To make the programming more smooth, defining these types are recommended:

Table 5-1 Overview of Typedefs

Type	Meaning
rmp_s8_t	Signed 8-bit integer. Example: typedef char rmp_s8_t;
rmp_s16_t	Signed 16-bit integer. Example: typedef short rmp_s16_t;
rmp_s32_t	Signed 32-bit integer. Example: typedef int rmp_s32_t;
rmp_u8_t	Unsigned 8-bit integer. Example: typedef unsigned char rmp_u8_t;
rmp_u16_t	Unsigned 16-bit integer. Example: typedef unsigned short rmp_u16_t;
rmp_u32_t	Unsigned 32-bit integer. Example:typedef unsigned int rmp_u32_t;

These three ones are required by RMP:

Table 5-2 Overview of Necessary Typedefs

Type	Meaning
rmp_ptr_t	The pointer integers' type. This shall be defined as an unsigned integer with the same length as one machine word. Example: typedef unsigned long rmp_ptr_t;
rmp_cnt_t	The counting variables' type. This shall be defined as an signed integer with the same length as one machine word. Example: typedef long rmp_cnt_t;
rmp_ret_t	The return values' type. This shall be defined as an signed integer with the same length as one machine word. Example: typedef long rmp_ret_t;

5.3.2 Defines

The second part is the porting of macro definitions. The macro definitions are listed as follows:

Table 5-3 Overview of Macro Defines

Name	Explanation
EXTERN	The extern keyword of the compiler. Some compiler dos not have standard extern keywords, this this define is necessary. Example: #define EXTERN extern
RMP_WORD_ORDER	The processor word length's[1] corresponding order. 32-bit processors will have a word order of 5, and 64-bit processors will have a word order of 6, etc. Example: #define RMP_WORD_ORDER 5
RMP_INIT_STACK	Initial stack start address. If the stack grows down, then this is the top of the stack; if the stack grows up, then this is the bottom of the stack. RMP declared two macros RMP_INIT_STACK_HEAD(X) and RMP_INIT_STACK_TAIL(X), which can help in making this macro. The former macro means that the address will be offset to higher addresses from the start address of the stack defined in the kernel, while the latter means that the address will be offset to lower addresses from the end address of the stack defined in the kernel. The unit of the offset is machine words. Example: #define RMP_INIT_STACK RMP_INIT_STACK_TAIL(16) The stack start address will be 16-word offset to lower address from the end of the initial stack.
RMP_INIT_STACK_SIZE	Initial thread stack size, in bytes. Example: #define RMP_INIT_STACK_SIZE 1024
RMP_MAX_PREEMPT_PRIO	The maximum number of preemptive priorities supported by the kernel. This number must be bigger than 3, and is recommended to

[1] In bits

Name	Explanation
	be a multiple of machine word length[1] if memory is not at a premium. Usually it suffices to define this as the same length with the machine word. Example: #define RMP_MAX_PREEMPT_PRIO 32
RMP_MAX_SLICES	The maximum timeslice value for thread or delays allowed by the kernel. Example: #define RMP_MAX_SLICES 100000
RMP_SEM_MAX_NUM	The maximum number of tokens allowed for a counting semaphore. Example: #define RMP_SEM_MAX_NUM 100
RMP_USE_HOOKS	Indicates whether the hook function is used. If yes, then the user is expected to provide the implementation of all four hook functions mentioned in section 2.5.3. Example: #define RMP_USE_HOOKS RMP_TRUE
RMP_MASK_INT() RMP_UNMASK_INT()	Indicates whether the system interrupts should be masked when the scheduler is locked. If this functionality is not desired, the two macros can be defined as empty values, and the interrupts which call the interrupt send functions may occur when the scheduler is locked. However, their interrupt send functions will fail due to the scheduler lock. If this functionality is desired, the RMP_MASK_INT() can be defined to mask all interrupts that may call interrupt send functions, and RMP_UNMASK_INT() can be defined to unmask these interrupts, thus the interrupt sending functions will not be called when the scheduler is locked. Some processors does not support disabling the interrupts below a certain priority, in this case the two macros can be configured to disable and enable global interrupts. In this case, the real-time performance will be affected. Example:

[1] In bits

Name	Explanation
	#define RMP_MASK_INT MASK(SYSPRIO) #define RMP_UNMASK_INT MASK(0x00) Refer to the Cortex-M3 port of RMP for more details. MASK(SYSPRIO) means masking all the interrupts below the SYSPRIO value, and MASK(0x00) means clearing the mask. The SYSPRIO value should be equal to the maximum interrupt priority level that can call interrupt send functions.

5.3.3 Low-Level Assembly Functions

RMP only requires 4 assembly snippet functions which can be implemented in either assembly or inline assembly. The names and explanations for these functions are as follows:

Table 5-4 Overview of Low-Level Assembly Functions

Function Name	Explanation
RMP_Disable_Int	Disable all processor interrupts.
RMP_Enable_Int	Enable all processor interrupts.
_RMP_Yield	Trigger the context switch.
_RMP_Start	Start the initial thread.

The implementation details and requirements will be explained later.

5.3.4 System Interrupt Vectors

RMP requires that the 2 vectors are written in assembly. The names and explanations are:

Table 5-5 Overview of System Interrupt Vectors

Vector Name	Explanation
System Timer Vector	Process system timer interrupt and manage timeslice usage.
Thread Context Switch Vector	Process thread context switches.

The implementation details and requirements will be explained in later sections.

5.3.5 Other Low-Level Functions

These low-level functions are used in booting or debugging of RMP. These functions can be implemented in either assembly or C. The list of these functions are as follows:

Table 5-6 Overview of Other Low-Level Functions

Function Name	Explanation
RMP_Putchar	Print a character to the kernel debugging console.
RMP_MSB_Get	Get the MSB position of the word.
_RMP_Low_Level_Init	Initialize the low-level hardware.
_RMP_Stack_Init	Initialize a thread's stack.

5.4 Porting of Assembly Functions

The detailed implementation requirements and prototypes of these functions are listed hereinafter.

5.4.1 Implementation of RMP_Disable_Int

This function just need to return immediately after disabling interrupts. There are no special precautions - in most cases it only involves a single instruction or register write.

Table 5-7 Implementation of RMP_Disable_Int

Prototype	void RMP_Disable_Int(void)
Explanation	Disable processor interrupts.
Return	None.
Parameter	None.

5.4.2 Implementation of RMP_Enable_Int

This function just need to return immediately after enabling interrupts. There are no special precautions - in most cases it only involves a single instruction or register write.

Table 5-8 Implementation of RMP_Enable_Int

Prototype	void RMP_Enable_Int(void)
Explanation	Enable processor interrupts.
Return	None.
Parameter	None.

5.4.3 Implementation of _RMP_Yield

This function needs to trigger the context switch interrupt vector. Usually it is writing to some memory address or executing some special instruction.

Table 5-9 Implementation of _RMP_Yield

Prototype	void _RMP_Yield(void)
Explanation	Software-triggerable context switch interrupt vector.
Return	None.
Parameter	None.

5.4.4 Implementation of _RMP_Start

This function implemented switch from kernel state to thread state, and is only called in the last step of the system booting process. After this, the system will start running. The only job for this function is to assign Stack to the stack pointer and then jump to the address specified by Entry. This function will never return.

Table 5-10 Implementation of _RMP_Start

Prototype	void _RMP_Start(rmp_ptr_t Entry, rmp_ptr_t Stack)
Explanation	Begin to execute the initial thread.
Return	None.
Parameter	rmp_ptr_t Entry Entry of the initial thread, which is RMP_Init.
	rmp_ptr_t Stack Stack address of the initial thread.

5.5 Porting of System Interrupt Vectors

RMP system only requires porting of two vectors, namely the system timer interrupt routine and the thread context switch routine. The timer interrupt vector may be implemented in pure C language, however the thread switch interrupt vector must be written in either assembly or inline assembly.

5.5.1 System Tick Timer Interrupt Vector

In the timer interrupt, only the following system tick timer interrupt processing function needs to be called:

Table 5-11 System Tick Timer Interrupt Processing Function

Prototype	void _RMP_Tick_Handler(rmp_ptr_t Ticks)
Explanation	Execute timer interrupt handling.
Return	None.
Parameter	rmp_ptr_t Ticks The number of ticks passed between the last interrupt and the current interrupt.

This function is implemented by the system and does not need to be implemented by the user.

5.5.2 Thread Context Switch Interrupt Vector

The context switch interrupt vector must be implemented in assembly, and must complete the following in the sequence listed below:

1. Switch to kernel stack and push all CPU registers to stack;
2. Call RMP_Save_Ctx save extra context such as FPU registers;
3. Place the current stack pointer into RMP_Cur_SP;
4. Call _RMP_Get_High_Rdy to pick the thread;
5. Assign RMP_Cur_SP to the current stack pointer;
6. Call RMP_Load_Ctx to load extra context such as FPU registers.
7. Pop all CPU registers from stack and switch to user stack, then exit interrupt.

RMP_Save_Ctx and RMP_Load_Ctx is introduced in chapter 2 and thus not introduced again here. Only the thread context switch interrupt processing function will be explained in detail.

Table 5-12 Thread Context Switch Interrupt Processing Function

Prototype	void _RMP_Get_High_Rdy(void)
Explanation	Do the context switch processing. This will update the RMP_Cur_SP and RMP_Cur_Thd, which is used by context switch assembly and other system calls.
Return	None.
Parameter	None.

This function is implemented by the system and the user does not need to implement it.

5.6 Porting of Other Low-Level Functions

Among all the low-level functions, RMP_Putchar and RMP_MSB_Get is discussed in chapter 2 and thus not discussed here. We just discuss _RMP_Low_Level_Init and _RMP_Stack_Init.

5.6.1 Low-Level Hardware Initialization

This function will initialize all low-level hardware. When initializing the interrupt system, it is necessary to set the system tick timer's interrupt priority to the lowest, and make the context switch interrupt's priority the second lowest, and they can't be nested with any other interrupt vectors. All vectors that may call RMP_Thd_Snd_ISR and RMP_Sem_Post_ISR may not nest with other vectors as well. For all other vectors such restriction does not apply and can be nested with each other freely.

Table 5-13 Implementation of _RMP_Low_Level_Init

Prototype	void _RMP_Low_Level_Init(void)
Explanation	Initialize basic hardware including PLL, CPU, interrupt controller and system tick timers, etc.
Return	None.
Parameter	None.

5.6.2 Initialization of Thread Stack

This function will be called by the thread creation system call to initialize a thread's stack. Because the second half of thread context switch will pop registers out of the stack, thus the registers needs to be placed in the corresponding order, especially the entry and arguments. The particular sequence depends on the processor and context switch vector implementation.

Table 5-14 Implementation of _RMP_Stack_Init

Prototype	void _RMP_Stack_Init(rmp_ptr_t Entry, rmp_ptr_t Stack, rmp_ptr_t Arg)
Explanation	Fill in the thread's stack and construct an interrupt return stack.
Return	None.
Parameter	rmp_ptr_t Entry The entry of the thread.
	rmp_ptr_t Stack The initial stack address of the thread.
	rmp_ptr_t Arg The parameter to pass to the argument.

5.7 Bibliography

None

Chapter 6 Appendix

6.1 Implementation of Special Kernel Functionality

In RMP, there are some features that are not implemented in the kernel by default. However, it is possible to implement these functionality in RMP; the detailed implementation mechanisms are listed below, and the user need to implement them if such functionality is desired.

6.1.1 Implementation of a Tick-Less Kernel

Tickless kernels usurally require a high-resolution time which is used to generate interrupts. To implement tick-less systems, we only needed to check the current thread's slices left and the earliest timer to expire, and set the timer to the smallest of these two in the system's timer tick hook RMP_Timer_Hook and the context switch hook RMP_Sched_Hook. The RMP provided a function _RMP_Get_Near_Ticks to get this value. When tickless feature is not implemented, there's no need to call the function. Should the time interval be longer than what the system tick timer can provide, the tick can be sliced into multiple ones until the last portion can be completed by a single timer delay.

Table 6-1 Getting the Time to the Nearest Ticks

Prototype	rmp_ptr_t _RMP_Get_Near_Ticks(void)
Explanation	Get the time to the nearest tick, which can be used to set the system's tick timer.
Return	rmp_ptr_t The number of ticks until the next interrupt.
Parameter	None.

This function is implemented by the system and the user does not need to implement it.

6.1.2 Saving and Restoring FPU Registers

To save and restore Floating-point Processing Unit (FPU) registers, the RMP_Save_Ctx and RMP_Load_Ctx hook should be used. In the save function, detect whether the task uses FPU first, if yes, then save the FPU registers to stack or some other position; in the load function, after confirming that the task uses FPU, just restore all the registers from where they are stored. In RMP, the recommended scheme is to only allow one thread to access the FPU - in this case the FPU context does not need to be saved or restored, thus boosting real-time responsiveness and efficiency.

For some architectures (Cortex-M, etc.), RMP has implemented floating point support by default, thus the aforementioned hook usage is unnecessary.

6.1.3 Thread Memory Protection

The memory access control that are pervasive in MCUs are Memory Protection Units (MPUs). MPUs usually have a set of registers where you can fill the access permissions of different memory regions. Each thread may have a different MPU configuration, and can limit its access permissions in a tailored way. A recommended implementation is to treat the MPU registers as coprocessor registers, and save/restore them in RMP_Save_Ctx and RMP_Load_Ctx. It is worth noting that the MPU configuration of each thread should allow accesses to RMP kernel memory and execution of RMP kernel code; this is because RMP does not have genuine user space and kernel space separation, and all system calls are actually implemented as function calls. This also explains the fact that RMP-based memory protection can only prevent the thread from executing random code or running into a fault, but cannot provide any information security.

6.1.4 Low-Power Design Considerations

In low-power designs, it is recommended to use tick-less implementations, and in the meantime, some instructions that can put the processor into sleep mode can be inserted into the RMP_Init_Idle hook[1].

6.2 Factors That are Known to Hamper Real-Time Performance in RMP

There are only two factors in RMP that can possibly hamper real-time performance, as listed below. The two factors are usually easy to avoid in real-world applications; generally speaking, if the system's real-timer performance is hampered due to these factors, the application is ill-designed.

6.2.1 Delay Queue

RMP employs a delay queue to manage all its delays[2]. From the head to the tail, the time to overflow always increases. Thus, at each clock tick, RMP only needs to check the first timer to see any of the timers can possibly overflow. However, this design requires that the list be traversed to find the correct position every time a timer is to be installed. This traversal keeps the incremental time property of the queue, but it also makes all insertion O(n) relative to the

[1] On ARM processors, WFI or WFE instructions should do the job, and on MSP430 some STATUS register modifications are needed to put the processor to sleep.

[2] That is, RMP_Thd_Delay

existing number of timers in the queue. It can be mathematically proved that there does not exist an algorithm that makes all operations on timers O(1), thus it is required that the application does not excessively use timers or block with timeout.

6.2.2 Lock Scheduler and Disabling Interrupts

RMP provides interfaces to lock the scheduler and disable interrupts. However, this obviously hampers the real-time performance of the system, thus it is best to avoid them in user applications. For instance, if it is known that only one thread will use a particular memory pool, then there's no need to lock the scheduler when allocating and freeing memory.

6.3 Reducing Memory Footprint of RMP

When memory is at a premium, it is necessary to carefully configure RMP to reduce ROM and ROM usage. This situation usually involves 16-bit MCUs such as MSP430 and low-end 32-bit MCUs such as Cortex-M0. In reality, the footprint of RMP can be shrunk by the following practices:

6.3.1 Lower the Number of Priorities Supported by the System

Different from other systems, RMP's thread mailboxes are integrated into threads at zero cost, while other functionalities will not consume any memory. Thus, there is only one compilation option that will dictate kernel RAM usage, which is the number of priorities supported by the RMP[1]. Each priority of RMP is managed by a doubly linked list, thus each priority will consume 2 machine words of RAM. On some small systems, there is no need to have too many priorities, and it suffices to configure the number to 3. This can save as much RAM as possible and squeeze the footprint of RMP kernel to less than 130 bytes[2].

6.3.2 Reconfigure the Manufacturer Libraries

The startup file and HAL provided by some manufacturers may statically allocate some heap and stack[3]. In cases where RMP is used, the manufacturer heap is completely unnecessary and should be set to 0; the manufacturer allocated stack, on the other hand, will be used by the OS kernel in interrupt responses[4], thus we only need to confirm the maximum stack usage of the interrupt handlers, and set the stack to a little more than this number[5]. Moreover, the HAL files

[1] That is, RMP_MAX_PREEMPT_PRIO

[2] If interrupt stack is not included, the actual size is 66 bytes

[3] I.e. the first few lines of the startup files of Cortex-M MCUs

[4] That is, used as a kernel stack

[5] In most cases a few dozen more bytes are sufficient

that are not used in the project should not be included, as this also help reducing RAM and ROM usage. If the HAL is too large to be tolerated, you can directly manipulate the peripheral registers instead.

6.3.3 Adjust Compiler Options

Adjusting compiler options may also help reducing ROM and RAM usage. The code size optimization compiler option is especially helpful with regards to reducing ROM usage, and it also helps with RAM usage in some cases.

6.3.4 Never use Dynamic Memory Management

RMP has a built-in dynamic memory allocator. This allocator is not recommended on tiny devices, because the allocator has some memory overhead itself. On tiny devices, the recommended practise is to allocate all kernel objects and variables statically.

6.4 Bibliography

None