演进・远古・原核(五阶)

Mutatus・Protero・Prokaryon(R.V)

M5P1(Prokaryon) R4T1

Light-Weight RTOS (Rev.4 Typ. 1)

Technical Manual

System Features

1.Highly Usable

- Light-weight and ease of use as first-class requirements
- Easy to configure development environment and (especially) path
- Provides generic inter-thread communication interfaces
- Provides templates and example projects for various toolchains and IDEs

2.Real-time

- Fully preemptive hard-real-time kernel
- Threads with higher priority will preempt threads with lower priority
- Threads with the same priority will round-robin
- Most system services' time complexity being O(1)

3.Highly portable

- Least assembly required amongst all RTOSes
- Strictly ISO/ANSI C89 compliant, compatible to all compilers
- Follows MISRA C guideline where possible
- Least resource required amongst al RTOSes
- Can run as a guest OS on other OSes

4.Highly efficient

- Highly optimized algorithms
- Critical sections as short as possible
- Lightening-fast context switching

5.Highly reliable

- Provides argument assertion, guarantees no unexpected behaviors
- Minimal debugging/logging facilities provided
- Formally verified to reach IEC 61508 SIL4 or Orange book A1 or EAL7+
  *Work in progress

Contents

Chapter 1 Introduction

1.1 General Description

In embedded IoT systems, the 16-bit and 32-bit processors are becoming increasingl;y prevalent. In the meantime, the requirement of low development time and high system reliability become an important factor. Thus, it is necessary to employ light-weight RTOSes to implement simple IoT applications. The light-weight OS may be used as a guest OS to run on hypervisors as well. For light-weight OSes, there's no memory protection at most times, thus formal methods and verification are necessary.

RMP is a RTOS highlighting simplicity, usability and efficiency. It implemented a fixed-priority round robin scheduler, along with many features commonly found in RTOSes. It also features a simple memory management interface as a standalone module. To sum up, the OS can be deployed on MCUs with as little as 2kB ROM and 1kB RAM.

This manual explained the kernel APIs of RMP from the user's perspective. We will introduce some basic concepts about RTOSes before we discuss RMP and its APIs in detail.

1.1.1 Design Goal and Specs

The first-class design goal is to make RMP as simple and usable as possible. In addition to this, RMP must match other RTOSes in terms of resource consumption and efficiency. RMP is open source and can be downloaded free of charge.

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 (forensic and medical equipments), some special terms apply. These special terms will be different for each particular application.

1.1.3 Terms and Definitions

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

1.1.3.1 Operating System

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

1.1.3.2 Thread

Threads refers to a control flow that have one independent stack and can be scheduled independent of each other. There can be multiple threads in one process, and they share the same address space.

1.1.3.3 Static Allocation

All resource allocation are done at compile time.

1.1.3.4 Dynamic Allocation

At least a part of the resources can be allocated 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 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 Variable)

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 (@Amazon).

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

API : RMProkaron (@EDI).

Kernel services : RT-Thread (@EDI).

All other references will be listed in respective chapters.

1.2 Forewords

Operating system is a kind of basic software that is responsible for CPU, memory and device management. For real-time operating systems, all operations of the system must be predictable and always meet its deadline. Generally speaking, there are two kinds of real-time systems: the former being soft real-time systems, which meets its deadline in most cases; the latter being hard real-time systems, which meets its deadline in all cases. Practically all embedded systems can be split in half, one part being soft real-time and the other part being hard real-time. One example is the motor controller: the GUI part is soft real-time, an the motor control part is hard real-time.

RMP is a basic hard real-time system, which exhibits all the basic features of a RTOS. It should be deployed on 16-bit and 32-bit machines at most times, and it require a system tick timer. RMP does not have genuine kernel space and user space; however, it is possible to configure the MPU to protect some ranges of memory. The hardware abstraction layer of RMP includes some simple assembly, which must be modified when porting to other architectures. Porting usually involves system tick timer, context switching, interrupt management and coprocessor management. These types of systems can use a customized linker script; however this is not always necessary, except in the case of MPU protection.

In such 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 uses 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. Usually it also have primitive memory management support, which is based on SLAB and buddy system.

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

Typical such systems include RMProkaron, RT-Thread, FreeRTOS, uC/OS, Salvo and ChibiOS. It is usually easy to perform a complete formal verification on such systems, due to the limited lines of 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.

1.3 Performance and Specs of RTOSes

There are hundreds of 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 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 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 acceptable. When in evaluation, the system load will also influence the results, thus the system load must remain the same when making measurements.

1.3.1 Kernel Size

Kernel size is a very important aspect of RTOS. 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 size and read-write size. The read-only size includes the code and constant data segment, while the read-write size includes the modifiable data size. On flash-based MCUs, the read-only segments will consume flash, and the read-write segments will consume SRAM.[1]

These RTOSes are highly configurable, thus its kernel size is rarely a fxed number. Usually they are tied closely to the detailed configuration. Therefore, to measure the performance, you should measure the kernel size under the minimal kernel configuration, common kernel configuration, and maximal kernel configuration.[1]

Obtaining kernel size data is as simple as compiling the kernel with a compiler and then using a dedicated binary viewer (such as Objdump) 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 the RTOS system call. The worst execution time refers to the maximum length of execution time under the most unfavorable conditions. The worst case RTOS execution time (WCET) is usually achieved when the longest system call is made and a large number of cache misses and TLB misses occur. RTOSes generally disables interrupts in the execution of system calls; the worst execution time is usually the longest time in which the system disables interrupts, so the impact of WCET on the real-time properties is enormous.

The WCET can be divided into two categories: the first is the WCET of system calls, and the

other is the WCET of inter-thread synchronization.

To get the first type of WCET, before calling a system call, jot down the time stamp Ts at this time, and then after the end of the system call, read the timer to get the time stamp Te. Then, read the timer twice in a row, note the two timestamps, Tts and Tte, and obtain the extra cost of reading the timer as Tte-Tts. In this case, the execution time is Te-Ts-(Tte-Tts). Repeat this on all system calls, then the WCET will be largest measurement among all measurements.

To get the second type of WCET, jot down the timestamp Ts at the sending side of the communication mechanism, then at the receiving end of the communication mechanism, read the timer get the timestamp Te. The cost measurement for reading the timer is similar to the first-type WCET. The resulting Te-Ts-(Tte-Tts) is the execution time. Repeat this on all communication mechanisms calls, then the WCET will be largest measurement 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 average execution time, and its standard deviation (and sometimes we also use the range) is called the jitter.

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

1.3.2.1 Thread Context Switch Time

The time cost of switching from one thread to another. We use the following method to measure this. In the measurement, except for the method using Te-Ts, it is also possible to use the difference between two Ts divided by 2 (the same applies hereinafter). There are two cases of thread switching, one case is to switch between threads with the same priority, the other is to wake up a high-priority thread low-priority thread. [2]

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

Process 1 : Thread A	Process 1 : Thread B

Loop forever
{
> Read Ts;
Switch to thread B;
}
	Loop forever
	{
	Read Te;
	> Switch to thread A;
	}

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

Process 1 : Thread A	Process 1 : Thread B

Loop forever
{
> Read Ts;
Wake B up;
}
	Loop forever
	{
	Read Te;
	> Sleep forever;
	}

1.3.2.2 Inter-Thread Asynchronous Communication Time

Asynchronous communication time between two threads. We use the following method to measure this. 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]

Process 1 : Thread A	Process 1 : Thread B

Loop forever
{
> Read Ts;
Send to thread B;
}
	Loop forever
	{
	Read Te;
	> Receive from own mailbox;
	}

1.3.3 Interrupt Response Time (IRT), 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 TLB misses occur during interrupt processing. Interrupt response time is the most important indicator of the RTOS, and it can even be said 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 (cannot wait until the C handler starts execution because register and stack maintenance are also part of the interrupt response time), resulting in a time stamp Ts; read the timer at the first line of code of the interrupt processing thread to get a timestamp Te. The measurement of the timer read cost is the same as above. The resulting Te-Ts-(Tte-Tts) 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 very important. We often get a distribution when we measure the interrupt response time of the same system multiple times. The average of this distribution is the average interrupt response time, and its standard deviation (and sometimes we also use range) is called interrupt response time jitter.

For a RTOS, we usually wish the interrupt response time, the worst interrupt response time and jitter wo be as small as possible. Interrupt response time measurement is usually conducted as follows [1] [3]:

Kernel	Thread A

Hardware interrupt handler
{
> Read Ts;
Send to thread A from interrupt;
}
	Loop forever
	{
	Read Te;
	> 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 IO operations' completion. The worst-case realistic 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 the realistic IRT, we need some extra hardware to support that kind of 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. First of all, the FPGA sends a signal on its output pin. At this time, the high-resolution timer in the FPGA starts to count. When the FPGA receives the signal on its input pin, the high-resolution timer in the FPGA stops working. The resulting internal FPGA timer value is the realistic IRT. The 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 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]:

FPGA	System under test

Loop forever
{
> Send stimulus and start the timer;
Wait until receipt of the response;
Stop the timer;
}
	Loop forever
	{
	Receive signal from I/O;
	Minimal processing routine;
	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 simple function calls. All system calls in RMP are listed hereinafter.

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
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

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 commodity RTOS, including threads, thread delays, thread mailboxes and counting semaphores. More complicated communication facilities can be easily developed with existing facilities。

RMP is recommended for usage in systems where less than 32 priorities and 64 threads 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 system services are required, the RME (M7M1) is recommended.

2.1.1 Scheduler

The RMP kernel is a fixed priority round robin scheduler, which is the simplest one among all real-time schedulers. It employs preemptive scheduling between threads with different priorities, and employs round-robin between threads with the same priority. Different from other RTOSes, the RMP scheduler never disables interrupts, which guarantees the real-time properties of the system. To enhance the real-time performances, 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. 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 to recommended. Though RMP can be used with a MPU, secure memory protection mechanisms are still impossible; when used without a MPU, no memory protection of any kind is provided. It is necessary that some programming standard be adhered to when writing applications.

2.1.3 Application Updating and Application Modules

As RMP is a small system, thus 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.

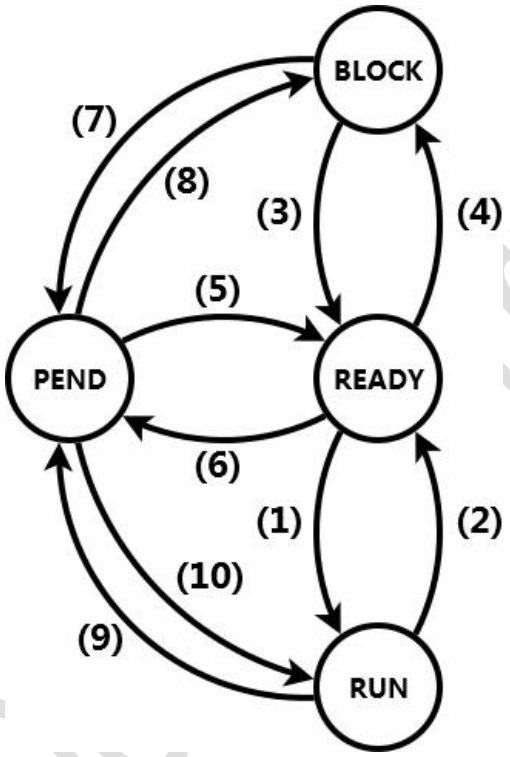
2.1.4 System Boot Sequence

The first thread of RMP is always RMP_Init, which is located 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 2.4 for detailed description.

2.2 Thread-Related Operations

As the whole RMP-based system run in the same address space, the RMP operating system only provided threads as the abstraction for control flow. There can be four states for each thread, namely READY, RUN, PEND and BLOCK (which includes four variants: blocked on delay/semaphore/send/receive). The BLOCK state can be set to return after some time 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:



The number markings have the meanings as follows:

No.	Meaning
(1)	It is the thread with the highest priority so far thus will be scheduled.
(2)	There's one thread with higher priority thus the current thread will go into READY state.
(3)	The thread is unblocked and with highest priority, thus it will go into RUN state.
(4)	The thread blocks and go into BLOCK state.
(5)	The thread is unblocked but not with highest priority, thus it will go into READY state.
(6)	The thread is pended while being READY and will go into the PEND state.
(7)	The thread is pended while BLOCK and will go into the PEND state.
(8)	The thread is unpended and still blocking, thus it will go into the PEND state.
(9)	The thread is pended while RUN and will go into the PEND state.
(10)	The thread is unpended and with the highest priority, thus it will go into the RUN state.

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

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 thread will still be scheduled.

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

2.2.2 Create a Thread

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.

Prototype	ret_t RMP_Thd_Crt(volatile struct RMP_Thd* Thread, ptr_t Entry, ptr_t Stack, ptr_t Arg, ptr_t Prio, ptr_t Slices)
Return	ret_t If successful, 0. If failed, one the following values will be returned:
	RMP_ERR_PRIO: The priority is not smaller thanRMP_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.
	ptr_t Entry The entry of the thread.
	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.
	ptr_t Arg The argument to pass to the thread. ptr_t Prio The priority of the thread. ptr_t Slices The number of timeslices for the thread. The unit of timeslice is ticks.

2.2.3 Delete a Thread

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 failure will be returned.

Prototype	ret_t RMP_Thd_Del(volatile struct RMP_Thd* Thread)
Return	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 being used.
Parameter	volatile struct RMP_Thd* Thread A pointer to the thread control block of the thread to be deleted.

2.2.4 Set 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.

Prototype	ret_t RMP_Thd_Set(volatile struct RMP_Thd* Thread, ptr_t Prio, ptr_t Slices)
Return	ret_t 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 not used.
Parameter	volatile struct RMP_Thd* Thread A pointer to the thread control block of the thread to be set parameters.
	ptr_t Prio The priority to set to the thread.
	ptr_t Slices The timeslice to set to the thread.

2.2.5 Suspend a Thread

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 way. 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.

Prototype	ret_t RMP_Thd_Suspend(volatile struct RMP_Thd* Thread)
Return	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 Resume a Thread

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

Prototype	ret_t RMP_Thd_Resume(volatile struct RMP_Thd* Thread)
Return	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 Delay a Thread

This operation will delay the current thread for some time.

Prototype	ret_t RMP_Thd_Delay(ptr_t Slices)
Return	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	ptr_t Slices The number of timeslices to delay.

2.2.8 Cancel a Thread's Delay

This operation will cancel a thread's delay operation.

Prototype	ret_t RMP_Thd_Cancel(volatile struct RMP_Thd* Thread)
Return	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. It should be noted that for all communication interfaces, when a number of threads blocked on it, the service sequence is always in chronological order, does not provide the priority queue cut function. 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 Send to a Thread's Mailbox

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

Prototype	ret_t RMP_Thd_Snd(volatile struct RMP_Thd* Thread, ptr_t Data, ptr_t Slices)
Return	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.
	ptr_t Data The data to send.
	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 Send to Mailbox from Interrupt

This operation will send to a thread from interrupt vector. Different from the version listed above, this version will always return without blocking.

Prototype	ret_t RMP_Thd_Snd_ISR(volatile struct RMP_Thd* Thread, ptr_t Data)
Return	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 or scheduler locked.
Parameter	volatile struct RMP_Thd* Thread A pointer to the thread control block of the thread to send to.
	ptr_t Data The data to send.

2.3.3 Receive from Thread Mailbox

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

Prototype	ret_t RMP_Thd_Rcv(ptr_t* Data, ptr_t Slices)
Return	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.
Parameter	ptr_t* Data This parameter is used for output, outputs the data received. 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 receive operation is complete.

2.3.4 Create a Semaphore

This operation will create a new counting semaphore.

Prototype	ret_t RMP_Sem_Crt(volatile struct RMP_Sem* Semaphore, ptr_t Number)
Return	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.
	ptr_t Number The initial semaphore number, must be smaller than RMP_SEM_MAX_NUM.

2.3.5 Delete a Semaphore

This operation will delete a semaphore. If one thread is blocked on it, then semaphore pend failure will be returned for it.

Prototype	ret_t RMP_Sem_Del(volatile struct RMP_Sem* Semaphore)
Return	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 Pend for a Semaphore

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

Prototype	ret_t RMP_Sem_Pend(volatile struct RMP_Sem* Semaphore, ptr_t Slices)
Return	ret_t If successful, the current number of semaphores will be returned. 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.
	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 pend operation is complete, or the semaphore is deleted.

2.3.7 Abort Pend on a Semaphore

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

Prototype	ret_t RMP_Sem_Abort(volatile struct RMP_Thd* Thread)
Return	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 Post to a Semaphore

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

Prototype	ret_t RMP_Sem_Post(volatile struct RMP_Sem* Semaphore, ptr_t Number)
Return	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.
	ptr_t Number The number of tokens to post.

2.3.9 Post to a Semaphore from Interrupt

This operation will post a certain number of tokens to a semaphore from the interrupt handler.

Prototype	ret_t RMP_Sem_Post_ISR(volatile struct RMP_Sem* Semaphore, ptr_t Number)
Return	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, or the scheduler is locked.
Parameter	volatile struct RMP_Sem* Semaphore A pointer to the semaphore control block of the semaphore to post to.
	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 effective on real-time systems. 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 4096 machine words (for 16-bit machines the pool shall not be smaller than 8kB and for 32-bit machines 16kB), and smaller than 128MB. Because RMP is mainly used in systems where memory is less than 128kB, this memory limit is rarely exceeded. 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 (the threads can use their own private pools or share them), thus the RMP's memory operations do not lock the scheduler by default. If locking the scheduler during operation is 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 3 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.

Prototype	ret_t RMP_Mem_Init(volatile void* Pool, ptr_t Size)
Return	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.
	ptr_t Size The size of the memory pool.

2.4.2 Allocate Memory

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.

Prototype	void* RMP_Malloc(volatile void* Pool, 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.
	ptr_t Size

	The memory to allocate, in bytes.

2.4.3 Free Memory

This operation will return memory to the 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 came from.

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.

2.5 Other System Interfaces

Other interfaces provided by RMP includes interrupt enable/disable, scheduler lock/unlock 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 RMP never 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. It is worth noting that this function pair does not have nesting count facilities.

The scheduler interface included a pair of functions that can lock and 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. Only disable interrupts where necessary, and the section should be kept as short as possible.

2.5.1.1 Disable 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 capability should be supplied by the user where necessary.

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

2.5.1.2 Enable Interrupts

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

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

2.5.1.3 Lock Scheduler

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

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

2.5.1.4 Unlock Scheduler

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

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 Clear Memory

This operation zeros a segment of memory.

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

2.5.2.2 Print a Character

This operation prints a character to the console (usually a serial interface).

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

2.5.2.3 Print a Signed Integer

This operation will print a signed integer to the console.

Prototype	cnt_t RMP_Print_Int(cnt_t Int)
Return	cnt_t

	The number of characters printed.
Parameter	cnt_t Int The integer to print.

2.5.2.4 Print a Unsigned Integer

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

Prototype	cnt_t RMP_Print_Uint(ptr_t Uint)
Return	cnt_t The number of characters printed.
Parameter	ptr_t Uint The unsigned integer to print.

2.5.2.5 Print a String

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

Prototype	cnt_t RMP_Print_String(s8* String)
Return	cnt_t The number of characters printed.
Parameter	s8* String The string to print.

2.5.2.6 Get the Most Significant Bit

This operation will get a word's most significant bit's (MSB) position (for instance, if the processor is 32-bit, it will return 0-31). If the number is zero, -1 (for 32-bit machines, 0xFFFFFFFF)will be returned.

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

2.5.2.7 Get the Least Significant Bit

This operation will get a word's least significant bit's (LSB) position (for instance, if the processor is 32-bit, it will return 0-31). If the number is zero, then a negative number will be returned.

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

2.5.2.8 Reverse the Bits

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.

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

2.5.2.9 Create a List

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

Prototype	void RMP_List_Crt(volatile struct RMP_List* Head)
Return	None.
Parameter	volatile struct RMP_List* Head The pointer to the list head structure to initialize.

2.5.2.10 Delete a Node in a List

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

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 Insert a Node in a List

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

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.

2.5.3 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.

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 (peripheral state or FPU registers), 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.

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 (peripheral state or FPU registers), 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.

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

2.5.3.4 Tick Timer Hook

This hook function will be called whenever there is a system tick. This function have a parameter which is a pointer to the number of ticks passed in the last tick. Thus, this function is capable of both reading and modifying the ticks passed from the last tick, and this can be used to implement a tickless system. Refer to chapter 4 for implementation details.

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

2.5.3.5 Init Thread Initial 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. Additionally, it is prohibited to use any other system calls that will cause a context switch or block in this hook.

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

2.5.3.6 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. Additionally, it is prohibited to use any other system calls that will cause a context switch or block in this hook.

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

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 Formal Verification

3.1 Introduction to Formal Verification

The definitive characteristic of embedded real-time operating systems is 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 that the delivery of software is on time.

Traditional software design techniques based on natural language thinking, designing and description. They are often 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 (sometimes automatic or structured) use case testing. 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 (EAL7+ is also inluded as a separate level)[1]:

EAL Level	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). 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 analysable) 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 OSes (for instance, RT-Thread, 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 (it have no memory protection anyway, thus CC/CAPP/LSPP standards are not applicable), it will be easy to certify it to a very high safety level.

3.2 Formal Model of the System

In the RMP system, the architecture-dependent part of the system is less than 50 source lines of code (SLOC), 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, it is futile to verify them one by one. Thus, the bulk of the verification work is on the 1500 SLOC of kernel.c file.

Future work in progress.

3.3 Formal Proof

Future work in progress.

3.4 Other Documents

Future work in progress.

Bibliography

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

Chapter 4 Porting RMP to New Architectures

4.1 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 will void.

4.2 Checklist Before Porting

4.2.1 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 (such as some PIC microcontrollers); the stack must be implemented in software (so that the SP and stack content can be software manipulated).

4.2.2 Compiler

RMP requires that the compiler is C89 compliant, and can produce code according tosome function calling convention. Because the RMP is written with strict accordance to the standard, and did not use any C runtime library components (it doesn't even require a startup routine). Thus, if the compiler is C89 compliant, the porting will be possible. Common compilers such as GCC, Clang/LLVM, MSVC, ARMCC, ICC, EWXXX(IAR), TASKING satisfy the requirement very well. Also, RMP did not use compiler extensions that vary across compiler implementations, such as bitfield, enum amd struct packing, thus maximum compatibility is guaranteed.

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. Do not use any aggressive optimization features (especially the experimental ones). On common compilers, the recommended optimization level is (GCC) -O2 or its equivalent optimization level.

4.2.3 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 conforms to the calling convention, and call C functions as well. Usually these are satisfied by default. If the compiler supports inline

assembly, then an standalone assembler is not required.

4.2.4 Debugger

RMP does not have any special requirements regarding the assembler. 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.

4.3 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.

4.3.1 Typedefs

For each architecture/compiler, the typedefs are always among the first things to be ported. It is worth noting that for some architectures/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:

Type	Meaning
s8	Signed 8-bit integer. Example: typedef char s8;
s16	Signed 16-bit integer. Example: typedef short s16;
s32	Signed 32-bit integer. Example: typedef int s32;
u8	Unsigned 8-bit integer. Example: typedef unsigned char u8;
u16	Unsigned 16-bit integer. Example: typedef unsigned short u16;
u32	Unsigned 32-bit integer. Example:typedef unsigned int u32;

These three ones are required by RMP:

Type	Meaning
ptr_t	The pointer integer ’s type. This shall be defined as an unsigned integer with the same length as one machine word. Example: typedef ptr_t unsigned long;
cnt_t	The counting variable’s type. This shall be defined as an signed integer with the same length as one machine word. Example: typedef cnt_t long;
ret_t	The return value’s type. This shall be defined as an signed integer with the same length as one machine word. Example: typedef ret_t long;

4.3.2 Defines

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

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 (in bits) 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 define. 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.
RMP_MAX_PREEMPT_PRIO	The maximum number of preemptive priorities supported by the kernel. This must be a multiple of machine word length (in bits). Usually it is sufficient 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 2.4.3. 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/enable global interrupts. In this case, the real-time performance will be affected. Example: #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.

4.3.3 Low-Level Assembly

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:

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.

4.3.4 System Interrupt Vectors

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

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

The implementation details and requirements will be explained later.

4.3.5 Other Low-Level Functions

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

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 hardwares.
_RMP_Stack_Init	Initialize a thread's stack.

4.4 Porting of Assembly Functions

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

4.4.1 Implementation of RMP_Disable_Int

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

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.

4.4.2 Implementation of RMP_Enable_Int

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

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.

4.4.3 Implementation of _RMP_Yield

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

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

4.4.4 Implementation of _RMP_Start

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

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.

4.5 Porting of System Interrupt Vectors

RMP system only needs to port two vectors, namely the system timer interrupt routine and the thread context switch routine. The timer interrupt vector can be implemented in pure C language, however the thread switch interrupt vector must be written in either assembly or inline assembly.

4.5.1 System Tick Timer Interrupt Vector

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

Prototype	void _RMP_Tick_Handler(ptr_t Ticks)
Explanation	Execute timer interrupt handling.
Return	None.
Parameter	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.

4.5.2 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 thread switch function will be explained in detail.

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.

4.6 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.

4.6.1 Low-Level Hardware Initialization

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.

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.

4.6.2 Initialization of Thread Stack

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

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.

4.6.3 Implementation of 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 and the context switch 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.

Prototype	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	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.

4.6.4 Saving and Restoring FPU Registers

To save and restore FPU registers, the RMP_Save_Ctx and RMP_Load_Ctx hook should be used. In the save function, the first job is to detect whether the task uses FPU; if yes, then save the FPU registers to stack or some other position; in the load function, the work is generally the same, after confirming that the task uses FPU, just restore all the registers from where they are stored.

4.6.5 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. On ARM architectures, WFI or WFE instructions should do the job, and on MSP430 some status register modifications are needed.

Bibliography

None