


PROKARON

# M5P01 (Prokaron) R6T1 Light-Weight Real-Time Operating System (Sixth Edition, Type One)

## Technical Reference Manual

Version R6T1 · June 2024

EDI-Systems · www.rtospro.com

# System Features

M5P01 (Prokaron) R6T1

## Light-Weight Real-Time Operating System (Sixth Edition, Type One)

Technical Reference Manual

### High usability

Light-weight and easy to use as first-class requirements. Easy to configure the development environment and the project paths. Provides the most common inter-thread communication interfaces.

### Hard real-time

A fully preemptive hard-real-time kernel. Round-robin scheduling between threads of the same priority. All key operations have O(1) time complexity.

### High portability

The least assembly language among all RTOSes. Written to ANSI C89 and MISRA C requirements for compiler compatibility. Extremely low resource requirements for a minimal system. Can run as a guest operating system on another operating system.

### High efficiency

Performance-critical code is carefully optimised. Extremely short thread switch time and interrupt response time.

### High reliability

Provides parameter assertion and pervasive interface-parameter checking. Formally verified to the equivalent of IEC 61508 SIL 4.[1]

[1] This work is not complete. The current software is equivalent to a SIL 2 pre-certification level.

# Contents


System Features ii

List of Tables viii

List of Figures ix

Revision History x

1 Overview 1

1.1 Introduction . . . 1

1.1.1 Design goals and specifications . . . 1

1.1.2 Copyright and license . . . 1

1.1.3 Terms and definitions . . . 1

1.1.4 Major reference systems . . . 2

1.2 Foreword . . . 2

1.3 Performance and specifications of RTOSes . . . 3

1.3.1 Kernel size . . . 3

1.3.2 Execution time, worst-case execution time, and jitter . . . 3

1.3.3 Interrupt response time, worst-case response time, and jitter . . . 4

1.3.4 Realistic interrupt response time, worst case, and jitter . . . 5

1.4 RMP system call format . . . 5

1.5 Chapter bibliography . . . 6

2 System Kernel 7

2.1 Kernel introduction . . . 7

2.1.1 Kernel scheduler . . . 7

2.1.2 Memory management and memory protection . . . 7

2.1.3 Application updating and application modules . . . 7

2.1.4 System boot sequence . . . 7

2.2 Thread scheduling interface . . . 8

2.2.1 Create a thread . . . 10

2.2.2 Delete a thread . . . 10

2.2.3 Set thread attributes . . . 11

2.2.4 Yield the processor . . . 11

2.2.5 Suspend a thread . . . 12

2.2.6 Resume a suspended thread . . . 12

2.2.7 Delay a thread . . . 13

2.2.8 Cancel a thread delay . . . 13

2.2.9 Thread busy-wait . . . 13

2.2.10 Peek at the next thread from an ISR . . . 14

2.3 Thread mailbox and semaphore interface . . . 14
2.3.1 Send to a thread mailbox . . . 14
2.3.2 Send to a thread mailbox from an interrupt . . . 16
2.3.3 Receive from a thread mailbox . . . 16
2.3.4 Create a semaphore . . . 16
2.3.5 Delete a semaphore . . . 17
2.3.6 Post to a semaphore . . . 17
2.3.7 Post to a semaphore from an interrupt . . . 18
2.3.8 Broadcast a semaphore . . . 18
2.3.9 Broadcast a semaphore from an interrupt . . . 18
2.3.10 Wait on a semaphore . . . 19
2.3.11 Wait on a semaphore and unlock the scheduler . . . 19
2.3.12 Abort a semaphore wait . . . 20
2.3.13 Query a semaphore count . . . 20
2.4 Memory management interface . . . 20
2.4.1 Initialise a memory pool . . . 21
2.4.2 Allocate from a memory pool . . . 21
2.4.3 Return memory to a memory pool . . . 22
2.4.4 Resize a memory block . . . 22
2.5 Hook function interface . . . 23
2.5.1 Mandatory hook interface . . . 23
2.5.2 Optional hook interface . . . 24
2.6 Auxiliary system interface . . . 27
2.6.1 Interrupt and scheduler interface . . . 27
2.6.2 Auxiliary library function interface . . . 28
2.7 Chapter bibliography . . . 32
3 Kernel Extension Interfaces 33
3.1 Kernel extension interface introduction . . . 33
3.1.1 Queue extension interface . . . 33
3.1.2 Soft timer extension interface . . . 34
3.2 First-in-first-out queue extension interface . . . 34
3.2.1 Create a FIFO . . . 35
3.2.2 Delete a FIFO . . . 35
3.2.3 Read from a FIFO . . . 36
3.2.4 Write to a FIFO . . . 36
3.2.5 Write to a FIFO from an interrupt . . . 36
3.2.6 Query the FIFO length . . . 37
3.3 Message queue extension interface . . . 37
3.3.1 Create a message queue . . . 38
3.3.2 Delete a message queue . . . 38
3.3.3 Send to a message queue . . . 38
3.3.4 Send to a message queue from an interrupt . . . 39
3.3.5 Receive from a message queue . . . 39
3.3.6 Query the message queue length . . . 40
3.4 Blocking message queue extension interface . . . 40
3.4.1 Create a blocking message queue . . . 40
3.4.2 Delete a blocking message queue . . . 41

3.4.3 Send to a blocking message queue . . . 41
3.4.4 Send to a blocking message queue from an interrupt . . . 42
3.4.5 Receive from a blocking message queue . . . 42
3.4.6 Query the blocking message queue length . . . 43
3.5 Soft timer extension interface . . . 43
3.5.1 Create an alarm manager . . . 44
3.5.2 Delete an alarm manager . . . 44
3.5.3 Dismantle an alarm manager . . . 44
3.5.4 Maintain an alarm manager . . . 45
3.5.5 Query the number of alarms . . . 45
3.5.6 Initialise an alarm . . . 46
3.5.7 Register an alarm . . . 47
3.5.8 Cancel an alarm . . . 47
3.5.9 Trigger an alarm . . . 48
3.6 Chapter bibliography . . . 48

**4 Lightweight Graphics Interface 49**
4.1 Introduction to embedded graphical interfaces . . . 49
4.1.1 Simplicity . . . 49
4.1.2 Diversity . . . 49
4.1.3 Planarity . . . 49
4.1.4 Restriction . . . 49
4.2 Enabling and configuring the graphics interface . . . 49
4.3 Basic drawing functions . . . 50
4.3.1 Draw a line . . . 50
4.3.2 Draw a dotted line . . . 51
4.3.3 Draw a rectangle . . . 52
4.3.4 Draw a rounded rectangle . . . 52
4.3.5 Draw a circle . . . 53
4.3.6 Draw a monochrome bitmap . . . 53
4.4 Anti-aliased drawing functions . . . 54
4.5 Widget drawing functions . . . 55
4.5.1 Draw a cursor . . . 55
4.5.2 Draw a checkbox . . . 55
4.5.3 Check a checkbox . . . 56
4.5.4 Clear a checkbox . . . 56
4.5.5 Draw a command button . . . 57
4.5.6 Press a command button . . . 57
4.5.7 Release a command button . . . 58
4.5.8 Draw a text-edit box . . . 58
4.5.9 Clear part of a text-edit box . . . 59
4.5.10 Draw a radio button . . . 59
4.5.11 Select a radio button . . . 60
4.5.12 Clear a radio button . . . 60
4.5.13 Draw a progress bar . . . 61
4.5.14 Change the progress of a progress bar . . . 61
4.6 GUI design guidelines . . . 62
4.6.1 Planar design . . . 62
4.6.2 Use a simple background and monochrome bitmaps . . . 62

4.6.3 Font-library design and implementation . . . 62
4.6.4 Design and implementation of complex controls . . . 63
4.7 Optimised implementation of low-level drawing . . . 63
4.7.1 Region control . . . 63
4.7.2 Macro inlining . . . 63
4.7.3 Collinear auto-increment . . . 63
4.7.4 Write combining . . . 63
4.7.5 Block buffering . . . 64
4.7.6 Palette method . . . 64
4.7.7 DMA transfer . . . 64
4.8 Chapter bibliography . . . 65

5 Formal Verification 66
5.1 Introduction to formal verification . . . 66
5.2 Formal specification of the system . . . 68
5.3 Formal proof . . . 68
5.4 Other documents . . . 68
5.5 Chapter bibliography . . . 68

6 Porting RMP to a New Architecture 69
6.1 Porting overview . . . 69
6.2 Checks before porting . . . 69
6.2.1 Processor . . . 69
6.2.2 Compiler . . . 69
6.2.3 Assembler . . . 70
6.2.4 Debugger . . . 70
6.3 Architecture-dependent parts of RMP . . . 70
6.3.1 Type definitions . . . 70
6.3.2 Macro definitions . . . 70
6.3.3 Assembly functions and macros . . . 74
6.3.4 System interrupt vectors . . . 74
6.3.5 Other low-level functions . . . 74
6.4 Porting the assembly functions and macros . . . 75
6.4.1 Implementation of RMP_Int_Disable . . . 75
6.4.2 Implementation of RMP_Int_Enable . . . 75
6.4.3 Implementation of _RMP_Start . . . 76
6.4.4 Implementation of RMP_YIELD . . . 76
6.4.5 Implementation of RMP_YIELD_ISR . . . 77
6.5 Porting the system interrupt vectors . . . 77
6.5.1 System tick vector . . . 77
6.5.2 Context-switch vector . . . 78
6.6 Porting the other low-level functions . . . 78
6.6.1 Low-level hardware initialisation . . . 78
6.6.2 Thread stack initialisation . . . 79
6.7 Chapter bibliography . . . 80

7 Appendix 81
7.1 Implementation of special kernel features . . . 81
7.1.1 Implementation of a tickless kernel . . . 81

7.1.2 Saving and restoring thread context . . . 83
7.1.3 Saving and restoring coprocessor context . . . 86
7.1.4 Implementation of thread memory protection. . . . 86
7.1.5 Low-power design considerations . . . 87
7.2 Known factors that affect real-time performance in RMP . . . 87
7.2.1 The delay queue. . . . 87
7.2.2 Deleting kernel objects . . . 87
7.2.3 Locking the scheduler and disabling interrupts . . . 87
7.3 Reducing the interrupt latency of RMP . . . 87
7.3.1 Masking system interrupts by priority. . . . 88
7.3.2 Masking system interrupts by source . . . 88
7.3.3 Delegating operations to a software interrupt . . . 88
7.3.4 Delegating operations to the timer interrupt . . . 89
7.3.5 Limiting interference from non-critical interrupts on critical interrupts . . . 90
7.4 Reducing the memory footprint of RMP. . . . 90
7.4.1 Reducing the number of preemption priorities . . . 90
7.4.2 Adjusting the vendor-provided library . . . 90
7.4.3 Adjusting compiler options . . . 91
7.4.4 Not using dynamic memory management . . . 91
7.5 Recommended rules for porting . . . 91
7.5.1 Recommended rules for processor word length . . . 91
7.5.2 Rules for saving and restoring segment registers . . . 92
7.5.3 Recommended rules for memory models. . . . 92
7.5.4 Recommended rules for porting compatibility. . . . 94
7.6 Vendor library compatibility . . . 95
7.6.1 Interrupt enable/disable compatibility . . . 95
7.6.2 Delay-function compatibility . . . 95
7.6.3 Code-generator compatibility . . . 96
7.7 RMP's deviations from the C standard and MISRA. . . . 96
7.7.1 C undefined behaviour: type-punning pointer aliasing may exist and does not follow strict aliasing . . . 96
7.7.2 C undefined behaviour: a byte is always eight bits and a `char` is always one byte . . . 97
7.7.3 C undefined behaviour: a pointer's actual representation is always a machine-recognisable address . . . 97
7.7.4 C undefined behaviour: converting a pointer to an unsigned integer no smaller than a machine word always preserves its actual representation . . . 98
7.7.5 C undefined behaviour: all "exactly fitting" structures are automatically packed . . . 98
7.7.6 C undefined behaviour: external identifiers are extended to 31 characters . . . 99
7.7.7 MISRA-C deviations . . . 100
7.8 Chapter bibliography. . . . 101

# List of Tables


1-1 Thread switch, first case. . . . 4
1-2 Thread switch, second case. . . . 4
1-3 Asynchronous communication between threads. . . . 4
1-4 Interrupt response time. . . . 5
1-5 Realistic interrupt response time. . . . 5
1-6 RMP system calls. . . . 6

2-1 Meaning of the transitions in Figure 2-1. . . . 9
2-2 Thread scheduling interface summary. . . . 9
2-3 Possible return values of the thread scheduling interfaces. . . . 9
2-4 Thread mailbox and semaphore interface summary. . . . 15
2-5 Possible return values of the thread mailbox and semaphore interfaces. . . . 15
2-6 Memory management interface summary. . . . 21

3-1 FIFO extension interface summary. . . . 35
3-2 Possible return values of the FIFO extension interfaces. . . . 35
3-3 Message queue extension interface summary. . . . 37
3-4 Possible return values of the message queue extension interfaces. . . . 37
3-5 Blocking message queue extension interface summary. . . . 40
3-6 Possible return values of the blocking message queue extension interfaces. . . . 40
3-7 Soft timer extension interface summary. . . . 43
3-8 Possible return values of the soft timer extension interfaces. . . . 43

4-1 Macros required to use the built-in embedded graphics interface. . . . 50

5-1 EAL levels and their descriptions. . . . 67

6-1 Overview of typedefs. . . . 71
6-2 Overview of necessary typedefs. . . . 71
6-3 Overview of macro definitions. . . . 71
6-4 Overview of low-level assembly functions and macros. . . . 74
6-5 Overview of system interrupt vectors. . . . 75
6-6 Overview of other low-level functions. . . . 75

7-1 RMP deviations from the Required part of MISRA-C:2012. . . . 100
7-2 RMP deviations from the Advisory part of MISRA-C:2012. . . . 100

# List of Figures

2-1 Thread state transition diagram. . . . . 8

# Revision History

| Revision | Date | Description |
|---|---|---|
| R1T1 | 15 February 2018 | Initial release. |
| R2T1 | 12 March 2018 | Added the porting notes and the appendix. |
| R4T3 | 31 July 2018 | Corrected errors in the manual. |
| R4T3 | 1 September 2018 | Updated the manual to the new version. |
| R4T4 | 10 October 2018 | Removed the 128 MB limit of the memory management unit; used a better rounded-rectangle drawing algorithm. |
| R4T5 | 25 November 2018 | Removed the limit on the number of priorities; allowed more aggressive memory-usage optimisations. |
| R5T1 | 1 March 2020 | Added extra thread-communication interfaces beyond the basic ones. |
| R5T2 | 7 May 2023 | Added broadcast and synchronous-unlock modes to semaphores. |
| R6T1 | 26 June 2024 | Adjusted the placement and declarations of the hook macros and the GUI configuration-macro rules. |

# 1. Overview

## 1.1 Introduction

Small Internet-of-Things systems increasingly use 16- and 32-bit microcontrollers, while the requirements on development time and reliability continue to grow. A lightweight system is therefore needed to build simple and efficient IoT applications. At the same time, lightweight virtualisation is becoming more important, so a small operating system is also useful as a guest under another operating system. Because such systems usually have no memory protection unit, formal verification is needed to establish their reliability.

RMP is a simple, efficient, and easy-to-use fully preemptive real-time operating system (RTOS). It provides the features expected of a typical RTOS kernel: a fully preemptive static-priority round-robin scheduler, simple memory management, concise communication mechanisms, and hardware-specific optimisations. RMP is designed to run efficiently on a microcontroller with only 2 KiB of ROM and 1 KiB of RAM.

This manual describes the RMP kernel API from the user's point of view. It first reviews a few concepts of small real-time operating systems, then describes the features and APIs of RMP chapter by chapter.

### *1.1.1 Design goals and specifications*

The goal of RMP is to create a simple, easy-to-use, open-source RTOS kernel with the best real-time behaviour and reliability in its class, and with average execution efficiency comparable to mainstream kernels.

### *1.1.2 Copyright and license*

RMP is released under *The Unlicense*. Special cases are handled by special provisions, and the exact terms differ for each possible case.[1]

### *1.1.3 Terms and definitions*

The basic terms used in this manual are defined as follows.

#### 1.1.3.1 Operating system

The basic software that runs on a device and manages the processor, memory, and other hardware resources.

#### 1.1.3.2 Thread

A schedulable entity that has its own execution stack and instruction stream. A process may contain several threads that share the process address space.

#### 1.1.3.3 Static allocation

A resource-allocation scheme in which the allocation is decided at compile time.

[1] For example, security equipment and medical equipment.

#### 1.1.3.4 Dynamic allocation

A resource-allocation scheme in which the allocation can change while the system is running.

#### 1.1.3.5 Soft real-time

A real-time guarantee under which most operations complete within their deadlines, but a small fraction may occasionally finish late.

#### 1.1.3.6 Hard real-time

A real-time guarantee under which every operation must complete within its deadline.

#### 1.1.3.7 Constant real-time

A real-time guarantee under which every operation is O(1) in the user input and the system configuration, and every execution completes within a practically meaningful constant deadline. This is the strongest form of real-time guarantee.

#### 1.1.3.8 Value-constant real-time

A real-time guarantee under which every operation and response is O(1) while a particular value is unchanged, and every execution completes within a practically meaningful constant deadline.

### *1.1.4 Major reference systems*

The scheduler and API are partly modelled on FreeRTOS (RT Engineering Ltd. /Amazon). The formal-proof approach is informed by seL4 (2016 Data61/CSIRO). The references and source material for the other chapters are listed at the end of each chapter.

## 1.2 Foreword

An operating system is the basic software that runs on a device and manages the processor, memory, and other hardware resources. For a real-time operating system, every operation must be correct and timely, and its execution time must be predictable. There are two broad classes of real-time systems: soft real-time systems, where most responses meet their deadlines, and hard real-time systems, where every response must meet its deadline. In practice, few real-time applications or operating systems are purely soft or purely hard; most combine both. A common example is an LCD human-machine interface, which is soft real-time, combined with a motor-control loop, which is hard real-time.

RMP is a basic real-time system: the smallest system that exhibits the essential properties of a real-time system. Such systems generally run on high-end 8/16-bit machines and low-end 32-bit machines and need a system timer. They do not distinguish user mode from kernel mode, but an MMU or MPU can be configured to protect a region of memory. They need a small amount of architecture-specific assembly for context-stack switching. Porting to a new architecture always involves this assembly, and often also the system timer, stack switching, interrupt management, and coprocessor management. Such systems may use a custom linker script or may not; a custom script is usually required when memory protection is used.

In such a system a task is represented by a thread. A task function may be reentrant. Each task has its own execution stack. Task code and kernel code may be compiled together or separately. A task calls a system function with an ordinary function call; there is no system-call instruction or software interrupt gate.

Such systems have priorities, and generally implement preemption between different priorities and round-robin scheduling at the same priority. They have a basic memory-management scheme, usually based on a slab or buddy allocator.

Interrupts may be completely transparent to the operating system, in which case the operating system need not know that an interrupt occurred. If a context switch is needed in an interrupt, the stack-switch assembly must be inserted into that interrupt, and the interrupt entry and exit must be written in assembly.

Typical systems in this class include RMProkaron, RT-Thread, FreeRTOS, uC/OS, Salvo, and ChibiOS. Complete formal verification of such a system is usually feasible because the kernel is very small. Traditional software engineering can systematically test functionality and show that defects are unlikely; formal verification can prove, from a specification or a set of logical rules, that the system has certain properties, which amounts to proving that under the stated conditions the system cannot have those defects.

## 1.3 Performance and specifications of RTOSes

Hundreds of RTOSes exist, and hobbyist kernels are countless. Their performance varies widely. The metrics below are directly comparable only when the processor architecture, the compiler, and the compiler options are the same. With a different architecture, compiler, or option set, the numbers are indicative but not comparable. A recommended method is to use an industry-standard ARM or MIPS processor with GCC `-O2`, or the Chronos simulator. The system load can also affect the values, so they are comparable only at the same load level.

### *1.3.1 Kernel size*

Kernel size is an important metric for an RTOS, because RTOSes are deployed on devices with very limited memory. It is measured in two parts: the read-only section, which contains code and read-only data, and the data section, which contains read-write data. On a flash-based microcontroller, the read-only section consumes flash and the data section consumes SRAM [1].

An RTOS is highly configurable, so its size is not fixed; it depends on the selected configuration. The metric should therefore be reported for the smallest, a typical, and the largest configuration.

Kernel size is easy to obtain: compile the kernel, then use a binary viewer to inspect the size of each section.[2]

### *1.3.2 Execution time, worst-case execution time, and jitter*

Execution time is the time taken by an RTOS system call. Worst-case execution time (WCET) is the maximum execution time under the least favourable conditions. An RTOS usually reaches its WCET when it executes the longest system call and suffers many cache misses along the way. System calls normally disable interrupts, so the WCET is often the longest interval with interrupts disabled; its effect on real-time behaviour is therefore very large.

There are two kinds of WCET: the WCET of a kernel system call and the WCET of inter-thread synchronisation. To measure the first, record a timestamp $T_s$ before the call and $T_e$ after it. Call the timer twice more to obtain $T_{ts}$ and $T_{te}$, so the timer overhead is $T_{te} - T_{ts}$. The execution time is $T_e - T_s - (T_{te} - T_{ts})$. The WCET is the largest such value over all system-call tests.

To measure the second, record $T_s$ at the sender and $T_e$ at the receiver. The timer overhead is measured as above, and the execution time is again $T_e - T_s - (T_{te} - T_{ts})$. The WCET is the largest value over all communication tests.

Jitter is also important. Repeated measurements of the execution or communication time produce a distribution. Its mean is the average execution time, and its standard deviation[3] is the execution-time jitter.

[2] For example, `objdump`.
[3] Sometimes the range is used instead.

For an RTOS, execution time, WCET, and jitter are all better when smaller. Execution time can be divided into the following categories.

#### 1.3.2.1 Thread switch time

Thread switch time is the time to switch from one thread to another. It can be measured with the $T_e - T_s$ method or, in a two-thread loop, as half the difference between two successive $T_s$ values. There are two cases: a switch between threads of the same priority, and a low-priority thread waking a high-priority thread.

In the first case, assume the two threads have the same priority and that thread A has just been switched in from thread B.

**Table 1-1.** Thread switch, first case.

| Thread A | Thread B |
|---|---|
| forever {<br>  record $T_s$;<br>  switch to thread B;<br>} | forever {<br>  record $T_e$;<br>  switch to thread A;<br>} |

In the second case, thread B has the higher priority and thread A has just been switched in from B.

**Table 1-2.** Thread switch, second case.

| Thread A | Thread B |
|---|---|
| forever {<br>  record $T_s$;<br>  wake B;<br>} | forever {<br>  record $T_e$;<br>  sleep;<br>} |

#### 1.3.2.2 Asynchronous communication time

Asynchronous communication time is the time to send and receive an asynchronous signal between threads. Assume thread B is already blocked at the receiver, thread A sends, and B has higher priority than A.

**Table 1-3.** Asynchronous communication between threads.

| Thread A | Thread B |
|---|---|
| forever {<br>  record $T_s$;<br>  send to B;<br>} | forever {<br>  record $T_e$;<br>  receive from own mailbox;<br>} |

### *1.3.3 Interrupt response time, worst-case response time, and jitter*

Interrupt response time is the time from an interrupt request to the RTOS invoking the handler thread. Worst-case interrupt response time is its maximum under the least favourable conditions, typically with many cache and TLB misses during interrupt handling. This is the most important RTOS metric; one could say that every RTOS design decision revolves around it. It is the most direct measure of the RTOS's response to an external stimulus.

To measure the worst case, record $T_s$ in the first assembly instruction of the interrupt vector[4] and $T_e$ at the first line of the handler thread. The timer overhead is measured as above, and the response time is $T_e - T_s - (T_{te} - T_{ts})$. The worst case is the largest value over all interrupt tests. Jitter is the standard deviation[5] of the distribution.

Smaller response time, worst-case response time, and jitter are better. The measurement is usually arranged as follows.

**Table 1-4.** Interrupt response time.

| Kernel | Thread A |
|---|---|
| hardware interrupt vector {<br>  record $T_s$;<br>  signal endpoint P;<br>} | forever {<br>  record $T_e$;<br>  receive from endpoint P;<br>} |

### *1.3.4 Realistic interrupt response time, worst case, and jitter*

Realistic interrupt response time is the time from an external signal input to the corresponding I/O output. Its worst case includes the factors that affect the ordinary worst case plus the intrinsic latency of the CPU and I/O hardware.

Measuring it requires external hardware. For example, connect one FPGA pin to a CPU input pin and another to a CPU output pin. The FPGA emits a signal and starts its high-resolution timer; when it receives the output signal, it stops the timer. The elapsed time is the realistic response time. The worst case is the largest value over all tests.

For a hardware/software system, smaller realistic response time, worst case, and jitter are better. The realistic worst case is approximately the worst-case execution time plus the worst-case interrupt response time plus the intrinsic CPU/IO delay. If an I/O input arrives just as a system call begins, the hardware vector cannot run until the call finishes; the output is produced only after that. The measurement is arranged as follows.

**Table 1-5.** Realistic interrupt response time.

| FPGA or oscilloscope | System under test |
|---|---|
| forever {<br>  emit signal and start timer;<br>  receive signal;<br>  stop timer;<br>} | forever {<br>  receive signal from I/O;<br>  minimal internal processing;<br>  output signal to I/O;<br>} |

## 1.4 RMP system call format

A system call is the only way to use a service provided by the system. In a library-structured operating system such as RMP, a system call is an ordinary function call. Table 1-6 lists all RMP system-call interfaces.

[4] It cannot wait until a C function is entered, because register and stack maintenance are also part of the interrupt response time.
[5] Sometimes the range is used instead.

Table 1-6. RMP system calls.

| System call | No. | Meaning |
|---|---|---|
| `RMP_Thd_Yield` | 0 | Yield the current thread's processor. |
| `RMP_Thd_Crt` | 1 | Create a thread. |
| `RMP_Thd_Del` | 2 | Delete a thread. |
| `RMP_Thd_Set` | 3 | Set a thread's priority and time slice. |
| `RMP_Thd_Suspend` | 4 | Suspend a thread. |
| `RMP_Thd_Resume` | 5 | Resume a suspended thread. |
| `RMP_Thd_Delay` | 6 | Delay for a period. |
| `RMP_Thd_Cancel` | 7 | Cancel a thread's delay. |
| `RMP_Thd_Snd` | 8 | Send a value to a thread's mailbox. |
| `RMP_Thd_Snd_ISR` | 9 | Send to a mailbox from an interrupt. |
| `RMP_Thd_Rcv` | 10 | Receive from the calling thread's mailbox. |
| `RMP_Sem_Crt` | 11 | Create a counting semaphore. |
| `RMP_Sem_Del` | 12 | Delete a counting semaphore. |
| `RMP_Sem_Pend` | 13 | Try to acquire a semaphore. |
| `RMP_Sem_Abort` | 14 | Abort a thread's semaphore wait. |
| `RMP_Sem_Post` | 15 | Post to a semaphore. |
| `RMP_Sem_Post_ISR` | 16 | Post to a semaphore from an interrupt. |

## 1.5 Chapter 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," M.S. thesis, Temple University, 2013.

# 2. System Kernel

## 2.1 Kernel introduction

The RMP kernel provides the simplest functions that a usable real-time operating system must have: threads, thread delays, a mailbox mechanism, and a semaphore mechanism. If the user needs more complex mechanisms, they can be built from these basic mechanisms. The system also provides a lightweight graphical interface library, which consumes no memory when it is not used.[1]

The system is recommended for applications with fewer than 32 priorities and fewer than 64 threads. Theoretically, the number of threads and priorities is unlimited, but for efficiency and ease of use, applications beyond this complexity are not recommended. For more complex embedded applications, a higher-level system such as RME (M7M01) is recommended.

### 2.1.1 Kernel scheduler

The RMP scheduler is a fully preemptive fixed-priority round-robin scheduler. It uses round-robin scheduling among threads of the same priority and preemptive scheduling between different priorities. Unlike some RTOSes, on some architectures the RMP scheduler never disables interrupts, which preserves real-time behaviour. When the processor has CLZ, FFS, or similar instructions, RMP can use them to accelerate the highest-priority search.

RMP has no built-in direct support for multi-core systems.[2] Multi-core support is possible by compiling several RMP copies and running one copy on each processor as a partitioned operating system, similar to Barrelfish [1].

### 2.1.2 Memory management and memory protection

RMP provides a memory allocator based on the two-level segregated fit (TLSF) algorithm [2][3]. Among O(1) allocators, it has high space efficiency. Dynamic memory allocation is not recommended in small embedded systems, so the allocator should be used as little as possible. RMP has no true memory protection: the address spaces of threads are not isolated, so application programs must follow good programming practice.

### 2.1.3 Application updating and application modules

Because the system is relatively small, it does not support application updating or application modules. A complete image built with this system should normally not exceed 128 KiB, so the whole image can be updated. If application modules are required, the user must implement them.

### 2.1.4 System boot sequence

The first thread of the RMP kernel is always `RMP_Init`, which is in the kernel code. The init thread performs the initial system initialisation and low-power handling, and it can never stop. It exposes two

[1] No relevant macros are defined.
[2] And it never will in the future.

hooks, `RMP_Init_Hook` and `RMP_Init_Idle`, which bring user code into the kernel's execution. These two interfaces are described in Section 2.6.

## 2.2 Thread scheduling interface

Because the whole RMP system runs in one address space, the kernel provides only the thread abstraction. A thread can be ready, running, suspended, or blocked, and a blocked thread can have a timeout after which it returns automatically. The suspended state can be applied to any thread, in which case the thread is temporarily prevented from executing regardless of its other state.

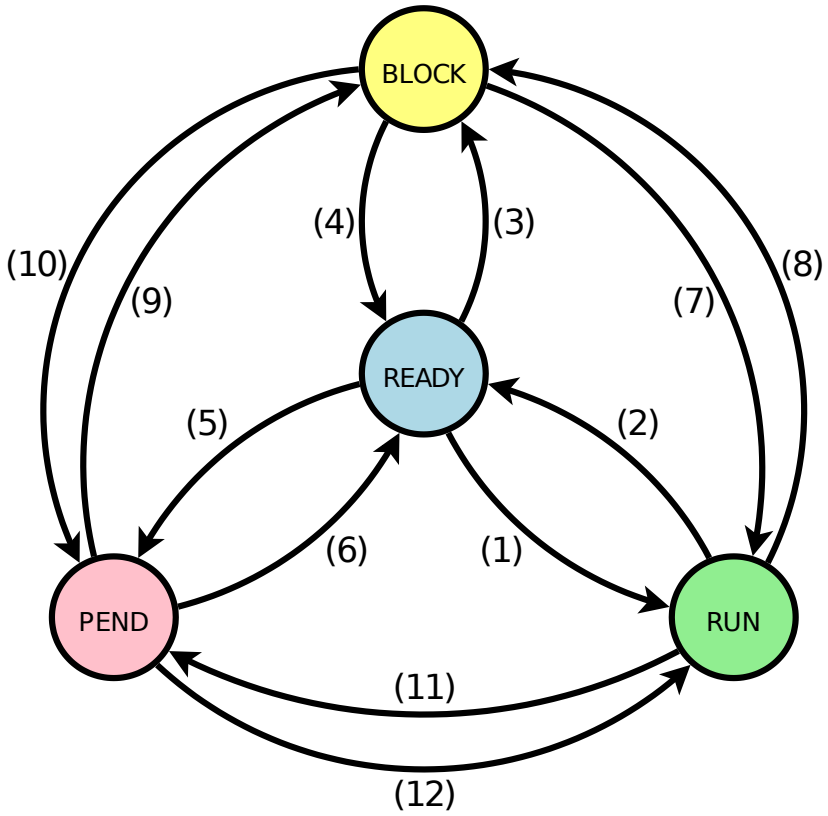



Figure 2-1. Thread state transition diagram.

The numbered transitions in Figure 2-1 are explained in Table 2-1. In the figure, PEND denotes the suspended state.

The blocking state includes four basic forms: delay, waiting on a semaphore, waiting to receive from a mailbox, and waiting to send to a mailbox.[3]

All thread scheduling interfaces are listed below.

[3] These are the four basic blocking forms in RMP.

Table 2-1. Meaning of the transitions in Figure 2-1.

| No. | Meaning |
|---|---|
| (1) | The thread is ready and has the highest priority, so it moves from ready to running. |
| (2) | The running thread is no longer the highest priority, so it moves from running to ready. |
| (3) | This possibility does not exist; a thread cannot go directly from ready to blocked. |
| (4) | The thread is unblocked and not suspended, and is not the highest priority, so it moves from blocked to ready. |
| (5) | The ready thread is suspended. |
| (6) | The suspended thread is resumed, is not blocked, and is not the highest priority, so it moves from suspended to ready. |
| (7) | The blocked thread is unblocked, is not suspended, and is the highest priority, so it moves from blocked to running. |
| (8) | The thread blocks on a mailbox, semaphore, or delay. |
| (9) | The suspended thread is resumed while still blocked, so it moves from suspended to blocked. |
| (10) | The blocked thread is suspended. |
| (11) | The running thread is suspended. |
| (12) | The suspended thread is resumed, is not blocked, and is the highest priority, so it moves from suspended to running. |

Table 2-2. Thread scheduling interface summary.

| Function | Purpose | Locked | ISR |
|---|---|---|---|
| `RMP_Thd_Crt` | Create a thread. | ✓ | ✗ |
| `RMP_Thd_Del` | Delete a thread. | ✓ | ✗ |
| `RMP_Thd_Set` | Set thread attributes. | ✓ | ✗ |
| `RMP_Thd_Yield` | Yield the processor. | ✗[4] | ✗ |
| `RMP_Thd_Suspend` | Suspend a thread. | ✗ | ✗ |
| `RMP_Thd_Resume` | Resume a thread. | ✓ | ✗ |
| `RMP_Thd_Delay` | Delay a thread. | ✗ | ✗ |
| `RMP_Thd_Cancel` | Cancel a thread delay. | ✓ | ✗ |
| `RMP_Thd_Loop` | Busy-wait loop. | ✓ | ✗ |

Table 2-3. Possible return values of the thread scheduling interfaces.

| Value | No. | Meaning |
|---|---|---|
| `RMP_ERR_THD` | −1 | Failure caused by the thread control block. |
| `RMP_ERR_PRIO` | −2 | Failure caused by the priority. |
| `RMP_ERR_SLICE` | −3 | Failure caused by the time slice. |
| `RMP_ERR_STACK` | −4 | Failure caused by the run stack. |
| `RMP_ERR_STATE` | −5 | Failure caused by the thread state. |
| `RMP_ERR_OPER` | −6 | Failure for any other reason. |

The exact meaning of a return code in a particular function is given in that function's entry.

[4] An operation that cannot block the current thread may be called with the scheduler locked; an operation that may block or yield the current thread must not. This criterion applies to the interfaces in later chapters as well.

### 2.2.1 Create a thread

#### RMP_Thd_Crt — Create a thread

```
rmp_ret_t RMP_Thd_Crt(volatile struct RMP_Thd* Thread, void* Entry, void* Param, void*
Stack, rmp_ptr_t Size, rmp_ptr_t Prio, rmp_ptr_t Slice)
```

This operation creates a new thread and makes it ready. The new thread is always inserted at the end of the time-slice round-robin queue of the same priority: if there is no other interference, the thread created first is always scheduled first.

Parameters

**`volatile struct RMP_Thd* Thread`**
Pointer to an empty thread control block to be used for this thread.

**`void* Entry`**
Entry address of the thread.

**`void* Param`**
Parameter passed to the thread.

**`void* Stack`**
Start address of the thread's run stack. If it is allocated as an array, fill in the first address of the array. Some architectures have special requirements for the stack memory address,[a] and the address must satisfy them.

**`rmp_ptr_t Size`**
Size of the thread's run stack in bytes. If it is allocated as an array, fill in the size of the array.

**`rmp_ptr_t Prio`**
Priority of the thread. In this system, the larger the number, the higher the priority.

**`rmp_ptr_t Slice`**
Number of time slices of the thread. The unit is clock ticks.

When `RMP_STKSEG_ENABLE` is non-zero, there is an additional `rmp_ptr_t Segment` parameter between `Param` and `Stack`, giving the thread's stack segment.

Returns

**`rmp_ret_t`**
0 on success. On failure, one of the following may be returned: `RMP_ERR_PRIO` if the priority is not less than `RMP_PREEMPT_PRIO_NUM`; `RMP_ERR_SLICE` if the slice is 0 or not less than `RMP_SLICE_MAX`; `RMP_ERR_STACK` if the stack area is 0 (NULL) or too small for the context; `RMP_ERR_THD` if the control block is 0 (NULL) or already in use.

[a] For example it must be located in the low 64 KiB of the address space. If a stack segment register is involved, see Section 7.5.3.2.

### 2.2.2 Delete a thread

#### RMP_Thd_Del — Delete a thread

```
rmp_ret_t RMP_Thd_Del(volatile struct RMP_Thd* Thread)
```

This operation deletes a thread in the system. After the thread is deleted, all resources are released, and the send waits of threads that are sending to it are released and return a send failure. Note that an RMP thread is not allowed to exit by returning. If any thread wants to terminate itself, it must

explicitly call this function to delete itself. This design is intentional, to explicitly remind the user that the thread is being deleted and to help the user avoid the "create–run once–delete" usage as much as possible.[a]

Parameters

**`volatile struct RMP_Thd* Thread`**
Pointer to the thread control block of the thread to delete.

Returns

**`rmp_ret_t`**
0 on success; `RMP_ERR_THD` if the control block is 0 (NULL) or unused.

[a] This usage should be replaced by "block–run once–block again" or thread-pool technology.

### 2.2.3 Set thread attributes

#### RMP_Thd_Set — Set thread attributes

```
rmp_ret_t RMP_Thd_Set(volatile struct RMP_Thd* Thread, rmp_ptr_t Prio, rmp_ptr_t Slice)
```

This operation sets the time slice and priority of a thread. When only one of them is to be set, simply ensure that the other parameter is `RMP_PREEMPT_PRIO_NUM` or `RMP_SLICE_MAX`.

Parameters

**`volatile struct RMP_Thd* Thread`**
Pointer to the thread control block of the thread whose priority and time slice are to be changed.

**`rmp_ptr_t Prio`**
Priority to set. Entering `RMP_PREEMPT_PRIO_NUM` means the priority is not changed. In this system, the larger the number, the higher the priority.

**`rmp_ptr_t Slice`**
Number of time slices to set. Entering `RMP_SLICE_MAX` means the time slice is not changed.

Returns

**`rmp_ret_t`**
0 on success. On failure: `RMP_ERR_PRIO` if the priority is greater than `RMP_PREEMPT_PRIO_NUM`; `RMP_ERR_SLICE` if the slice is 0 or greater than `RMP_SLICE_MAX`; `RMP_ERR_THD` if the control block is 0 (NULL) or unused.

### 2.2.4 Yield the processor

#### RMP_Thd_Yield — Yield the processor

```
void RMP_Thd_Yield(void)
```

This operation makes the current thread give up the CPU, and the scheduler automatically selects the next thread to schedule. If the current thread has the highest priority and is the only thread at that priority, it is still scheduled.

Parameters

**None**
No parameters.

#### Returns

**None**
No return value.

### 2.2.5 Suspend a thread

#### RMP_Thd_Suspend

Suspend a thread

```
rmp_ret_t RMP_Thd_Suspend(volatile struct RMP_Thd* Thread)
```

This operation suspends a thread, making the scheduler temporarily stop scheduling it. A thread can be suspended in any state, but its existing delay, send, receive, or other states continue to exist; that is, the suspended state is superimposed on one of the delay, send, receive, and other states. When the delay, send, receive, or other state ends, if the thread is still suspended, it is placed in the pure suspended state and still does not participate in scheduling until the suspended state is manually removed.

#### Parameters

**`volatile struct RMP_Thd* Thread`**
Pointer to the thread control block of the thread to suspend.

#### Returns

**`rmp_ret_t`**
0 on success. On failure: `RMP_ERR_THD` if the control block is 0 (NULL) or unused; `RMP_ERR_STATE` if the thread is already suspended and cannot be suspended again. This check is performed even when `RMP_CHECK_ENABLE` is disabled.

### 2.2.6 Resume a suspended thread

#### RMP_Thd_Resume

Resume a thread

```
rmp_ret_t RMP_Thd_Resume(volatile struct RMP_Thd* Thread)
```

This operation removes a thread's suspended state, giving it a chance to participate in scheduling again.

#### Parameters

**`volatile struct RMP_Thd* Thread`**
Pointer to the thread control block of the thread to resume.

#### Returns

**`rmp_ret_t`**
0 on success. On failure: `RMP_ERR_THD` if the control block is 0 (NULL) or unused; `RMP_ERR_STATE` if the thread is not suspended and cannot be resumed. This check is performed even when `RMP_CHECK_ENABLE` is disabled.

### 2.2.7 Delay a thread

#### RMP_Thd_Delay

Delay a thread

```
rmp_ret_t RMP_Thd_Delay(rmp_ptr_t Slice)
```

This operation delays the current thread for a period of time.

Parameters

**`rmp_ptr_t Slice`**
Time to delay, in time slices.

Returns

**`rmp_ret_t`**
0 on success. On failure: `RMP_ERR_SLICE` if the slice is 0 or not less than `RMP_SLICE_MAX`; `RMP_ERR_OPER` if the delay is cancelled before it completes. This check is performed even when `RMP_CHECK_ENABLE` is disabled.

### 2.2.8 Cancel a thread delay

#### RMP_Thd_Cancel

Cancel a thread delay

```
rmp_ret_t RMP_Thd_Cancel(volatile struct RMP_Thd* Thread)
```

This operation cancels a thread's delay.

Parameters

**`volatile struct RMP_Thd* Thread`**
Pointer to the thread control block of the thread whose delay is to be cancelled.

Returns

**`rmp_ret_t`**
0 on success. On failure: `RMP_ERR_THD` if the control block is 0 (NULL) or unused; `RMP_ERR_STATE` if the thread is not in the delay state and cannot be cancelled. This check is performed even when `RMP_CHECK_ENABLE` is disabled.

### 2.2.9 Thread busy-wait

#### RMP_Thd_Loop

Busy-wait a thread

```
void RMP_Thd_Loop(rmp_ptr_t Loop)
```

This operation makes the current thread enter a dead loop for a period of time. Unlike other thread interfaces, it can be used with the scheduler locked or interrupts disabled, and the delay granularity is fine, but it wastes processor performance.

Parameters

**`rmp_ptr_t Loop`**
Time to delay, in loop counts. The conversion between loop counts and processor cycles may differ with different processors, toolchains, and optimisation levels.

Returns

**None**
No return value.

### 2.2.10 Peek at the next thread from an ISR

#### RMP_Thd_Peek_ISR

Peek at the next thread from an ISR

```
volatile struct RMP_Thd* RMP_Thd_Peek_ISR(void)
```

This operation can be used in an interrupt or with the scheduler lock held to peek at the thread that will replace the current thread. If no event that can unblock a thread has occurred in the system, the thread that replaces the current thread is the thread returned by this function. As a special case, if the current thread is the highest-priority ready thread and there are other ready threads at that priority besides the current thread, this operation returns the next thread after the current thread, that is, the thread that would be selected if `RMP_Thd_Yield` were called at this moment. The main purpose of this operation is to periodically detect whether a high-priority thread is ready in a long critical section of a low-priority thread that uses the scheduler lock; if so, the low-priority thread can give up the scheduler lock early to reduce interrupt latency.

##### Parameters

**None**
No parameters.

##### Returns

**volatile struct RMP_Thd***
Pointer to the control block of the next thread.

## 2.3 Thread mailbox and semaphore interface

RMP provides simple and efficient inter-thread communication mechanisms, including thread mailboxes and semaphores. A semaphore has an operation count, queued waiting, blocking and unlocking, and broadcast wake-up, so it can serve as a condition variable, a counter, or a lock inside a thread. These two basic mechanisms can be used separately or combined into more complex communication interfaces. Every blocking communication interface has three options: return immediately after detecting that blocking is possible, return after a blocking timeout, or block forever. This greatly increases the flexibility of the interfaces. Note that when several threads are blocked on any communication interface, they are always served in chronological order; priority jumping is not supported. If a high-priority thread blocks after a low-priority thread, the low-priority thread is served first and the high-priority thread second.

All thread mailbox and semaphore interfaces are listed below.

They may return the following values.

### 2.3.1 Send to a thread mailbox

#### RMP_Thd_Snd

Send to a thread mailbox

```
rmp_ret_t RMP_Thd_Snd(volatile struct RMP_Thd* Thread, rmp_ptr_t Data, rmp_ptr_t Slice)
```

This operation sends a value to a thread's mailbox. If the mailbox is empty, the value is stored; if

Table 2-4. Thread mailbox and semaphore interface summary.

| Function | Purpose | Locked | ISR |
|---|---|---|---|
| `RMP_Thd_Snd` | Send to a thread mailbox. | ✗ | ✗ |
| `RMP_Thd_Snd_ISR` | Send to a thread mailbox from an interrupt. | ✗ | ✓ |
| `RMP_Thd_Rcv` | Receive from a thread mailbox. | ✗ | ✗ |
| `RMP_Sem_Crt` | Create a semaphore. | ✓ | ✗ |
| `RMP_Sem_Del` | Delete a semaphore. | ✓ | ✗ |
| `RMP_Sem_Post` | Post to a semaphore. | ✓ | ✗ |
| `RMP_Sem_Post_ISR` | Post to a semaphore from an interrupt. | ✗ | ✓ |
| `RMP_Sem_Bcst` | Broadcast a semaphore. | ✓ | ✗ |
| `RMP_Sem_Bcst_ISR` | Broadcast a semaphore from an interrupt. | ✗ | ✓ |
| `RMP_Sem_Pend` | Wait on a semaphore. | ✗ | ✗ |
| `RMP_Sem_Pend_Unlock` | Wait on a semaphore and unlock the scheduler. | ✓ (single lock) | ✗ |
| `RMP_Sem_Abort` | Abort a semaphore wait. | ✓ | ✗ |
| `RMP_Sem_Cnt` | Query a semaphore count. | ✓ | ✗ |

Table 2-5. Possible return values of the thread mailbox and semaphore interfaces.

| Value | No. | Meaning |
|---|---|---|
| `RMP_ERR_THD` | −1 | The thread control block caused the operation to fail. |
| `RMP_ERR_SLICE` | −3 | The time slice caused the operation to fail. |
| `RMP_ERR_STATE` | −5 | The thread state caused the operation to fail. |
| `RMP_ERR_OPER` | −6 | The operation failed for another reason. |
| `RMP_ERR_SEM` | −7 | The semaphore control block caused the operation to fail. |

the receiver is waiting, it is woken. If the mailbox is full, the sending thread blocks.

### Parameters

**`volatile struct RMP_Thd* Thread`**
Pointer to the thread control block of the receiving thread.

**`rmp_ptr_t Data`**
Value to send.

**`rmp_ptr_t Slice`**
Number of time slices to wait if the mailbox is full. 0 means return immediately if blocking would occur; a value from 0 to `RMP_SLICE_MAX` means wait at most that many slices; a value greater than or equal to `RMP_SLICE_MAX` means wait forever.

### Returns

**`rmp_ret_t`**
0 on success. On failure: `RMP_ERR_THD` if the control block is 0 (NULL) or unused; `RMP_ERR_OPER` if blocking would occur with a zero timeout, if the operation is a self-send, or if the wait timed out or the target thread was deleted. The last check is performed even when `RMP_CHECK_ENABLE` is disabled.

### 2.3.2 Send to a thread mailbox from an interrupt

#### RMP_Thd_Snd_ISR

Send to a thread mailbox from an ISR

```
rmp_ret_t RMP_Thd_Snd_ISR(volatile struct RMP_Thd* Thread, rmp_ptr_t Data)
```

This operation sends a value to a thread's mailbox from an interrupt vector. Unlike the ordinary send, it always returns immediately and does not block.

Parameters

**`volatile struct RMP_Thd* Thread`**
Pointer to the thread control block of the receiving thread.

**`rmp_ptr_t Data`**
Value to send.

Returns

**`rmp_ret_t`**
0 on success. On failure: `RMP_ERR_THD` if the control block is 0 (NULL) or unused; `RMP_ERR_OPER` if the mailbox is full. This check is performed even when `RMP_CHECK_ENABLE` is disabled.

### 2.3.3 Receive from a thread mailbox

#### RMP_Thd_Rcv

Receive from a thread mailbox

```
rmp_ret_t RMP_Thd_Rcv(rmp_ptr_t* Data, rmp_ptr_t Slice)
```

This operation receives a value from the current thread's mailbox. If the mailbox is empty, the receiving thread blocks.

Parameters

**`rmp_ptr_t* Data`**
Output pointer for the received value. If `RMP_NULL` is passed, the received value is discarded.

**`rmp_ptr_t Slice`**
Number of time slices to wait if the mailbox is empty. 0 means return immediately if blocking would occur; a value from 0 to `RMP_SLICE_MAX` means wait at most that many slices; a value greater than or equal to `RMP_SLICE_MAX` means wait forever.

Returns

**`rmp_ret_t`**
0 on success. On failure: `RMP_ERR_OPER` if blocking would occur with a zero timeout, or if the wait timed out. This check is performed even when `RMP_CHECK_ENABLE` is disabled.

### 2.3.4 Create a semaphore

#### RMP_Sem_Crt

Create a semaphore

```
rmp_ret_t RMP_Sem_Crt(volatile struct RMP_Sem* Semaphore, rmp_ptr_t Number)
```

This operation creates a counting semaphore.

Parameters

**volatile struct RMP_Sem* Semaphore**
Pointer to an empty semaphore control block to be used.

**rmp_ptr_t Number**
Initial count of the semaphore. It should be less than `RMP_SEM_CNT_MAX`.

#### Returns

**rmp_ret_t**
0 on success. On failure: `RMP_ERR_SEM` if the control block is 0 (NULL) or already used; `RMP_ERR_OPER` if the count is not less than `RMP_SEM_CNT_MAX`.

### 2.3.5 Delete a semaphore

## RMP_Sem_Del

Delete a semaphore

```
rmp_ret_t RMP_Sem_Del(volatile struct RMP_Sem* Semaphore)
```

This operation deletes a semaphore. If any thread is blocked on it, that thread's semaphore-wait function returns `RMP_ERR_OPER` directly.

#### Parameters

**volatile struct RMP_Sem* Semaphore**
Pointer to the semaphore control block to delete.

#### Returns

**rmp_ret_t**
0 on success; `RMP_ERR_SEM` if the control block is 0 (NULL) or unused.

### 2.3.6 Post to a semaphore

## RMP_Sem_Post

Post to a semaphore

```
rmp_ret_t RMP_Sem_Post(volatile struct RMP_Sem* Semaphore, rmp_ptr_t Number)
```

This operation posts a number of counts to a semaphore and wakes waiting threads while the count remains positive.

#### Parameters

**volatile struct RMP_Sem* Semaphore**
Pointer to the semaphore control block.

**rmp_ptr_t Number**
Number of counts to post. It must not be 0.

#### Returns

**rmp_ret_t**
0 on success. On failure: `RMP_ERR_SEM` if the control block is 0 (NULL) or unused; `RMP_ERR_OPER` if the number is 0 or the resulting count would reach `RMP_SEM_CNT_MAX`.

### 2.3.7 Post to a semaphore from an interrupt

#### RMP_Sem_Post_ISR

Post to a semaphore from an ISR

```
rmp_ret_t RMP_Sem_Post_ISR(volatile struct RMP_Sem* Semaphore, rmp_ptr_t Number)
```

This operation posts a number of counts to a semaphore from an interrupt vector. It never blocks.

**Parameters**

**`volatile struct RMP_Sem* Semaphore`**
Pointer to the semaphore control block.

**`rmp_ptr_t Number`**
Number of counts to post. It must not be 0.

**Returns**

**`rmp_ret_t`**
0 on success. On failure: `RMP_ERR_SEM` if the control block is 0 (NULL) or unused; `RMP_ERR_OPER` if the number is 0 or the resulting count would reach `RMP_SEM_CNT_MAX`.

### 2.3.8 Broadcast a semaphore

#### RMP_Sem_Bcst

Broadcast a semaphore

```
rmp_ret_t RMP_Sem_Bcst(volatile struct RMP_Sem* Semaphore)
```

This operation wakes all threads waiting on a semaphore and returns the number of threads woken. The semaphore count is not changed.

**Parameters**

**`volatile struct RMP_Sem* Semaphore`**
Pointer to the semaphore control block.

**Returns**

**`rmp_ret_t`**
Number of threads woken on success; `RMP_ERR_SEM` if the control block is 0 (NULL) or unused.

### 2.3.9 Broadcast a semaphore from an interrupt

#### RMP_Sem_Bcst_ISR

Broadcast a semaphore from an ISR

```
rmp_ret_t RMP_Sem_Bcst_ISR(volatile struct RMP_Sem* Semaphore)
```

This operation wakes all threads waiting on a semaphore from an interrupt vector and returns the number of threads woken.

**Parameters**

**`volatile struct RMP_Sem* Semaphore`**
Pointer to the semaphore control block.

**Returns**

**rmp_ret_t**
Number of threads woken on success; `RMP_ERR_SEM` if the control block is 0 (NULL) or unused.

### 2.3.10 Wait on a semaphore

#### RMP_Sem_Pend

Wait on a semaphore

```
rmp_ret_t RMP_Sem_Pend(volatile struct RMP_Sem* Semaphore, rmp_ptr_t Slice)
```

This operation tries to acquire a semaphore. If the count is positive, the count is decremented and the call returns the remaining count; otherwise the current thread blocks.

Parameters

**volatile struct RMP_Sem* Semaphore**
Pointer to the semaphore control block.

**rmp_ptr_t Slice**
Number of time slices to wait if the semaphore count is 0. 0 means return immediately if blocking would occur; a value from 0 to `RMP_SLICE_MAX` means wait at most that many slices; a value greater than or equal to `RMP_SLICE_MAX` means wait forever.

Returns

**rmp_ret_t**
On success, the remaining semaphore count when the semaphore was acquired immediately, or 0 when the thread was woken after blocking. On failure: `RMP_ERR_SEM` if the control block is 0 (NULL) or unused; `RMP_ERR_OPER` if blocking would occur with a zero timeout, or if the wait timed out, was aborted, or the semaphore was deleted. This check is performed even when `RMP_CHECK_ENABLE` is disabled.

### 2.3.11 Wait on a semaphore and unlock the scheduler

#### RMP_Sem_Pend_Unlock

Wait on a semaphore and unlock the scheduler

```
rmp_ret_t RMP_Sem_Pend_Unlock(volatile struct RMP_Sem* Semaphore, rmp_ptr_t Slice)
```

This operation tries to acquire a semaphore when the scheduler is already locked; it automatically unlocks the scheduler on exit.

Parameters

**volatile struct RMP_Sem* Semaphore**
Pointer to the semaphore control block.

**rmp_ptr_t Slice**
Number of time slices to wait if the semaphore count is 0.

Returns

**rmp_ret_t**
On success, the remaining count or 0 as for `RMP_Sem_Pend`. On failure: `RMP_ERR_SEM` if the control block is 0 (NULL) or unused; `RMP_ERR_OPER` if the scheduler lock count is not exactly one, if blocking would occur with a zero timeout, or if the wait timed out, was aborted, or the semaphore was deleted. The last two checks are performed even when `RMP_CHECK_ENABLE` is disabled. In the

lock-count error case the function still calls `RMP_Sched_Unlock` once, so the caller must account for that decrement and must not unlock twice.

### 2.3.12 Abort a semaphore wait

#### RMP_Sem_Abort

Abort a semaphore wait

```
rmp_ret_t RMP_Sem_Abort(volatile struct RMP_Thd* Thread)
```

This operation aborts a thread's wait for a semaphore.

**Parameters**

**`volatile struct RMP_Thd* Thread`**
Pointer to the thread control block whose semaphore wait is to be aborted.

**Returns**

**`rmp_ret_t`**
0 on success. On failure: `RMP_ERR_THD` if the control block is 0 (NULL) or unused; `RMP_ERR_STATE` if the thread is not waiting on a semaphore. This check is performed even when `RMP_CHECK_ENABLE` is disabled.

### 2.3.13 Query a semaphore count

#### RMP_Sem_Cnt

Query a semaphore count

```
rmp_ret_t RMP_Sem_Cnt(volatile struct RMP_Sem* Semaphore)
```

This operation queries the current count of a semaphore.

**Parameters**

**`volatile struct RMP_Sem* Semaphore`**
Pointer to the semaphore control block.

**Returns**

**`rmp_ret_t`**
Current semaphore count on success; `RMP_ERR_SEM` if the control block is 0 (NULL) or unused.

## 2.4 Memory management interface

RMP provides a memory allocator based on TLSF. It manages dynamic memory efficiently. TLSF is a two-level fit algorithm: memory blocks are first placed into first-level intervals (FLI) by powers of two, such as 127–64 bytes and 255–128 bytes, and within each interval they are placed into linear second-level intervals (SLI). For example, the 127–64 byte FLI has eight SLIs: [127,112], [111,96], ..., [79,64]. During allocation, the allocator first checks the next SLI for the requested size; if a block is available it is allocated, otherwise the nearest higher SLI is searched. When memory is freed, the block is merged immediately and placed into the corresponding SLI for the next allocation.

In RMP's TLSF implementation, the number of FLIs is determined by the pool size and the number of SLIs is fixed at 8. FLI 0 corresponds to 64–127 byte blocks, and a single pool must not be smaller than

1024 machine words.[5] The allocator is optimised for pools below 128 MiB and is not recommended for larger pools. The pool and its size must both be aligned to a machine word, and the minimum allocation is 64 bytes.

The relationship between pools and threads is very flexible: one-to-one, one-to-many, many-to-one, or many-to-many. To maximise performance in the single-thread case, the RMP memory interfaces do not provide multi-thread safety, which is very different from the other kernel interfaces. If multi-thread safety is needed, it must be implemented with the scheduler lock or a semaphore. Reads and writes outside an allocated region are forbidden; they can corrupt the allocator's data structures and cause exceptions.

Table 2-6. Memory management interface summary.

| Function | Purpose | Notes |
|---|---|---|
| `RMP_Mem_Init` | Initialise a memory pool. | The memory management interfaces provide no thread safety. Multi-thread mutual exclusion must be provided by the application, for example with the scheduler lock or a semaphore. |
| `RMP_Malloc` | Allocate from a memory pool. | |
| `RMP_Free` | Return memory to a memory pool. | |
| `RMP_Realloc` | Resize a memory block. | |

Apart from `RMP_NULL`, the only possible return value is `RMP_ERR_MEM` (−8).

### 2.4.1 Initialise a memory pool

#### RMP_Mem_Init — Initialise a memory pool

```
rmp_ret_t RMP_Mem_Init(volatile void* Pool, rmp_ptr_t Size)
```

This operation initialises the pool according to the passed memory-pool size and address. The pool size and address must be aligned to a machine word.

**Parameters**

**`volatile void* Pool`**
Start address of the memory pool.

**`rmp_ptr_t Size`**
Size of the memory pool.

**Returns**

**`rmp_ret_t`**
0 on success; `RMP_ERR_MEM` if the pool is 0 (NULL), smaller than 1024 machine words, unaligned, or overflows.

### 2.4.2 Allocate from a memory pool

#### RMP_Malloc — Allocate from a memory pool

```
void* RMP_Malloc(volatile void* Pool, rmp_ptr_t Size)
```

This operation tries to allocate memory from a memory pool. The minimum allocation is 64 bytes;

[5] For a 16-bit machine this is at least 2 KiB; for a 32-bit machine at least 4 KiB. Below this size there is no reason to use a pool.

if the requested size is smaller than that but greater than 0, 64 bytes are actually allocated; if the passed size is 0, allocation failure[a] is returned directly.

#### Parameters

**`volatile void* Pool`**
Memory pool from which to allocate.

**`rmp_ptr_t Size`**
Size of memory to allocate.

#### Returns

**`void*`**
Pointer to the allocated memory; `RMP_NULL` if no memory is available.

[a] This check is ignored when `RMP_CHECK_ENABLE` is disabled.

### 2.4.3 Return memory to a memory pool

## RMP_Free

Return memory to a memory pool

```
void RMP_Free(volatile void* Pool, void* Mem_Ptr)
```

This operation returns memory to a memory pool. `Mem_Ptr` must be an address returned by `RMP_Malloc` that has not yet been returned, and it must be returned to the pool from which it was allocated;[a] otherwise the operation fails silently. If `Mem_Ptr` is `RMP_NULL`, nothing is done.

#### Parameters

**`volatile void* Pool`**
Memory pool to return to.

**`void* Mem_Ptr`**
Pointer to the memory to return.

#### Returns

**`None`**
No return value.

[a] These checks are ignored when `RMP_CHECK_ENABLE` is disabled.

### 2.4.4 Resize a memory block

## RMP_Realloc

Resize a memory block

```
void* RMP_Realloc(volatile void* Pool, void* Mem_Ptr, rmp_ptr_t Size)
```

This operation tries to resize a memory block. `Mem_Ptr` must be an address returned by `RMP_Malloc` that has not yet been returned, and the passed pool must be the same as the one used when `RMP_Malloc` was called.[a] On success, it returns the original block or a new block whose first `Size` bytes have the same contents as the original address; when `Size` is greater than 0 and less than 64, 64 bytes are allocated by default. On failure it returns 0 (NULL) and the original block is unaffected. If `Size` is 0, the behaviour is the same as `RMP_Free` and the block is released directly.[b] If `Mem_Ptr` is `RMP_NULL`, the behaviour is the same as `RMP_Malloc`.

#### Parameters

**`volatile void* Pool`**
Memory pool.

**`void* Mem_Ptr`**
Memory block to resize.

**`rmp_ptr_t Size`**
New size.

#### Returns

**`void*`**
Pointer to the resized block; `RMP_NULL` on failure or when `Size` is 0.

[a] These checks are ignored when `RMP_CHECK_ENABLE` is disabled.
[b] This may differ from some implementations.

## 2.5 Hook function interface

To make it convenient to insert user-required functions at some key points in system operation, RMP also provides hook functions. Hook functions are divided into mandatory hooks and optional hooks: the former are declared as functions and must be provided; the latter are declared as macros and need not be filled in.

### *2.5.1 Mandatory hook interface*

The hook functions that must be implemented by the user are as follows.

#### 2.5.1.1 Idle-thread first-run hook

This hook function is called once when the idle thread first runs. Its typical use is to create other threads, semaphores, and message queues, or to perform some system initialisation work. Note that this hook executes with interrupts disabled throughout, so it must not call any operation that may depend on interrupts or any operation that enables interrupts. If such operations are needed, perform them in another thread.

##### RMP_Init_Hook — Idle-thread first-run hook

```
void RMP_Init_Hook(void)
```

Called once when the idle thread first runs.

#### Parameters

**`None`**
No parameters.

#### Returns

**`None`**
No return value.

#### 2.5.1.2 Idle-thread repeated-run hook

This hook function is called repeatedly when the idle thread runs. Its typical use is to enter a low-power mode to save energy, or to perform performance analysis and stack-check printing. The idle thread must not be put into a blocking state in this hook; otherwise the system will crash.

### RMP_Init_Idle

Idle-thread repeated-run hook

```
void RMP_Init_Idle(void)
```

Called repeatedly when the idle thread runs.

#### Parameters

**None**
No parameters.

#### Returns

**None**
No return value.

## 2.5.2 Optional hook interface

The optional hook macros that the user may choose to implement when needed are as follows.

### 2.5.2.1 System start hook

If this hook macro is defined, it is called immediately after the system completes processor initialisation. Because each port initialises only part of the most basic hardware at system startup, further hardware initialisation is recommended to be placed inside this hook. If the desired basic system configuration conflicts with the basic initialisation provided by the platform's configuration header,[6] the basic system configuration can also be redone inside this hook.

### RMP_START_HOOK

System start hook

```
void RMP_START_HOOK(void)
```

Called immediately after processor initialisation.

#### Parameters

**None**
No parameters.

#### Returns

**None**
No return value.

### 2.5.2.2 System context-save hook

If this hook macro is defined, it is called after the system completes basic context saving. If additional context[7] must be saved, it can be pushed onto the stack in this function. See Chapter 4 for the implementation method.

### RMP_CTX_SAVE

System context-save hook

```
void RMP_CTX_SAVE(void)
```

Called after basic context saving.

[6] For example, configuring the system to a completely different operating frequency.
[7] FPU and other peripheral register banks.

#### Parameters

**None**
No parameters.

#### Returns

**None**
No return value.

### 2.5.2.3 System context-restore hook

If this hook macro is defined, it is called before the system completes basic context restoration. If additional context[8] must be restored, it can be popped from the stack in this function. See Chapter 4 for the implementation method.

#### RMP_CTX_LOAD — System context-restore hook

```
void RMP_CTX_LOAD(void)
```

Called before basic context restoration.

#### Parameters

**None**
No parameters.

#### Returns

**None**
No return value.

### 2.5.2.4 Scheduler hook

If this hook macro is defined, it is called when the system completes selection of the current thread. If a tickless kernel is to be implemented, the relevant functions can be called in this hook. See Section 7.1.1 for the implementation of a tickless system.

#### RMP_SCHED_HOOK — Scheduler hook

```
void RMP_SCHED_HOOK(void)
```

Called after the current thread is selected.

#### Parameters

**None**
No parameters.

#### Returns

**None**
No return value.

### 2.5.2.5 System tick hook

If this hook macro is defined, it is called when the system tick arrives.

[8] FPU and other peripheral register banks.

### RMP_TIM_HOOK

System tick hook

```
void RMP_TIM_HOOK(rmp_ptr_t Slice)
```

Called when the system tick arrives.

#### Parameters

**`rmp_ptr_t Slice`**
The `Slice` parameter that `_RMP_Tim_Handler` receives when called.

#### Returns

**`None`**
No return value.

#### 2.5.2.6 Delay-insertion hook

If this hook macro is defined, it is called when a thread is inserted into the delay queue.

### RMP_DLY_HOOK

Delay-insertion hook

```
void RMP_DLY_HOOK(rmp_ptr_t Slice)
```

Called when a thread is inserted into the delay queue.

#### Parameters

**`rmp_ptr_t Slice`**
The `Slice` parameter that the delayed system function receives when called.

#### Returns

**`None`**
No return value.

#### 2.5.2.7 Error-logging hook

If this hook macro is defined, it is called when the system prints error information. If it is not defined, the system prints errors to the console by default.

### RMP_LOG

Error-logging hook

```
void RMP_LOG(const char* File, long Line, const char* Date, const char* Time)
```

Called when the system prints error information.

#### Parameters

**`const char* File`**
Name of the file where the error occurred.

**`long Line`**
Line number where the error occurred.

**`const char* Date`**
Compilation date.

**`const char* Time`**
Compilation time.

#### Returns

**None**
No return value.

## 2.6 Auxiliary system interface

The other auxiliary interfaces provided by RMP include interrupt enable/disable and scheduler lock/unlock. These interfaces provide convenience when the user writes an application.

### *2.6.1 Interrupt and scheduler interface*

In RMP, the interrupt-system interface consists only of a pair of functions that disable and enable interrupts. This pair is provided to the user because on some architectures RMP itself does not disable interrupts during operation. Unless absolutely necessary, disabling interrupts is not recommended, because it greatly affects the system's real-time performance.

The scheduler-system interface consists of a pair of functions that enable and disable the scheduler with a nesting count. This pair is also provided to the user. Because locking the scheduler also reduces real-time performance, frequent use is not recommended. The general policy is: if the problem can be solved without locking the scheduler, do not lock it; if both locking the scheduler and disabling interrupts can solve it, lock the scheduler; only when interrupts must be disabled should they be disabled, and the interrupt-disabled interval should be as short as possible.

#### 2.6.1.1 Disable interrupts

This operation disables the processor's response to all interrupt sources, including the system timer interrupt and the scheduling interrupt. This function has no nesting count; if nesting is required, the user must implement it.

#### RMP_Int_Disable — Disable interrupts

```
void RMP_Int_Disable(void)
```

Disables the processor's response to all interrupt sources.

#### Parameters

**None**
No parameters.

#### Returns

**None**
No return value.

#### 2.6.1.2 Enable interrupts

This operation enables the processor's response to all interrupt sources. This function has no nesting count; if nesting is required, the user must implement it.

#### RMP_Int_Enable — Enable interrupts

```
void RMP_Int_Enable(void)
```

Enables the processor's response to all interrupt sources.

#### Parameters

**None**
No parameters.

#### Returns

**None**
No return value.

#### 2.6.1.3 Lock the scheduler

This operation locks the scheduler, preventing it from selecting a new thread to schedule. This function has a nesting count.

### RMP_Sched_Lock — Lock the scheduler

```
void RMP_Sched_Lock(void)
```

Locks the scheduler.

#### Parameters

**None**
No parameters.

#### Returns

**None**
No return value.

#### 2.6.1.4 Unlock the scheduler

This operation unlocks the scheduler, allowing it to select a new thread to schedule again. This function has a nesting count.

### RMP_Sched_Unlock — Unlock the scheduler

```
void RMP_Sched_Unlock(void)
```

Unlocks the scheduler.

#### Parameters

**None**
No parameters.

#### Returns

**None**
No return value.

### *2.6.2 Auxiliary library function interface*

To make application writing convenient, RMP provides a series of library functions, including memory clearing, debug printing, and linked-list operations. They are listed below. The default logging function is `RMP_Log`, and the kernel coverage printer is `RMP_Cov_Print`.

#### 2.6.2.1 Clear memory

This operation clears a block of memory to zero.

### RMP_Clear — Clear memory

```
void RMP_Clear(volatile void* Addr, rmp_ptr_t Size)
```

Clears a block of memory to zero.

#### Parameters

**`volatile void* Addr`**
Start address of the memory block to clear.

**`rmp_ptr_t Size`**
Size of the memory block to clear, in bytes.

#### Returns

**`None`**
No return value.

#### 2.6.2.2 Print one character

This operation prints one character to the console.[9]

### RMP_Putchar — Print one character

```
void RMP_Putchar(char Char)
```

Prints one character to the console.

#### Parameters

**`char Char`**
The character to print.

#### Returns

**`None`**
No return value.

#### 2.6.2.3 Print a signed integer

This operation prints a signed integer of one machine word in signed decimal to the debug console.

### RMP_Int_Print — Print a signed integer

```
rmp_cnt_t RMP_Int_Print(rmp_cnt_t Int)
```

Prints a signed integer in signed decimal.

#### Parameters

**`rmp_cnt_t Int`**
The signed integer to print.

#### Returns

[9] Usually a serial port.

**rmp_cnt_t**
The number of characters printed.

#### 2.6.2.4 Print an unsigned integer

This operation prints an unsigned integer of one machine word in hexadecimal without a prefix to the debug console.

### RMP_Hex_Print

Print an unsigned integer

```
rmp_cnt_t RMP_Hex_Print(rmp_ptr_t Uint)
```

Prints an unsigned integer in hexadecimal without a prefix.

#### Parameters

**rmp_ptr_t Uint**
The unsigned integer to print.

#### Returns

**rmp_cnt_t**
The number of characters printed.

#### 2.6.2.5 Print a string

This operation prints a string of at most 255 characters to the debug console.

### RMP_Str_Print

Print a string

```
rmp_cnt_t RMP_Str_Print(const rmp_s8_t* String)
```

Prints a string to the debug console.

#### Parameters

**const rmp_s8_t* String**
The string to print.

#### Returns

**rmp_cnt_t**
The number of characters printed.

#### 2.6.2.6 Get the most significant bit of a word

This operation obtains the bit number of the most significant bit of a word.[10] If the number is 0, it must return −1.[11] This is a generic function and cannot use processor-specific acceleration instructions; if possible, use the `RMP_MSB_GET` macro.

### RMP_MSB_Generic

Get the most significant bit of a word

```
rmp_ptr_t RMP_MSB_Generic(rmp_ptr_t Value)
```

[10] For example a 32-bit processor returns 0–31.
[11] For example a 32-bit processor should return `0xFFFFFFFF`.

Obtains the bit number of the most significant bit of a word.

#### Parameters

**`rmp_ptr_t Value`**
The unsigned integer whose most-significant-bit number is to be found.

#### Returns

**`rmp_ptr_t`**
The bit number of the word's most significant bit.

### 2.6.2.7 Get the least significant bit of a word

This operation obtains the bit number of the least significant bit of a word.[12] If the number is 0, it returns a value equal to the processor word length.[13] This is a generic function and cannot use processor-specific acceleration instructions; if possible, use the `RMP_LSB_GET` macro.

#### RMP_LSB_Generic — Get the least significant bit of a word

```
rmp_ptr_t RMP_LSB_Generic(rmp_ptr_t Value)
```

Obtains the bit number of the least significant bit of a word.

#### Parameters

**`rmp_ptr_t Value`**
The unsigned integer whose least-significant-bit number is to be found.

#### Returns

**`rmp_ptr_t`**
The bit number of the word's least significant bit.

### 2.6.2.8 Create a doubly linked circular list

This operation initialises the head of a doubly linked circular list.

#### RMP_List_Crt — Create a doubly linked circular list

```
void RMP_List_Crt(volatile struct RMP_List* Head)
```

Initialises the head of a doubly linked circular list.

#### Parameters

**`volatile struct RMP_List* Head`**
Pointer to the list-head structure to initialise.

#### Returns

**`None`**
No return value.

### 2.6.2.9 Delete nodes from a doubly linked circular list

This operation deletes one or a series of nodes from a doubly linked list.

[12] For example a 32-bit processor returns 0–31.
[13] For example a 32-bit processor returns 32.

### RMP_List_Del

Delete nodes from a doubly linked circular list

```
void RMP_List_Del(volatile struct RMP_List* Prev, volatile struct RMP_List* Next)
```

Deletes one or a series of nodes from a doubly linked circular list.

#### Parameters

**`volatile struct RMP_List* Prev`**
Pointer to the predecessor of the node (or node group) to delete.

**`volatile struct RMP_List* Next`**
Pointer to the successor of the node (or node group) to delete.

#### Returns

**`None`**
No return value.

#### 2.6.2.10 Insert a node into a doubly linked circular list

This operation inserts one node into a doubly linked list.

### RMP_List_Ins

Insert a node into a doubly linked circular list

```
void RMP_List_Ins(volatile struct RMP_List* New, volatile struct RMP_List* Prev,
volatile struct RMP_List* Next)
```

Inserts one node into a doubly linked circular list.

#### Parameters

**`volatile struct RMP_List* New`**
Pointer to the new node to insert.

**`volatile struct RMP_List* Prev`**
Pointer to the predecessor of the insertion position.

**`volatile struct RMP_List* Next`**
Pointer to the successor of the insertion position.

#### Returns

**`None`**
No return value.

## 2.7 Chapter bibliography

None.

# 3. Kernel Extension Interfaces

The primitive kernel is small, atomic, and easy to verify, but it is too basic to use directly in many applications. RMP therefore provides a set of higher-level extension interfaces assembled from one or more primitive calls. They are slower than the primitives; for performance-critical paths, the application should use the primitives directly.

## 3.1 Kernel extension interface introduction

### *3.1.1 Queue extension interface*

For message queues, RMP extends the first-in-first-out queue (FIFO), the message queue, and the blocking message queue.

A FIFO is similar to a non-blocking pipe in Unix. It does not support blocking; it is only a thread-safe wrapper around a linked list and can pass data by polling. It is the fastest.

A message queue is similar to the mechanism of the same name in other operating systems. It supports multiple producers and multiple consumers; only the consumer can block, while the producer cannot, and the queue length is unlimited. It is a combined wrapper of a counting semaphore and a FIFO, and its speed is medium.

A blocking message queue supports a queue-length limit and producer blocking in addition to the message queue, so that the producer can be slowed down when the consumer cannot consume all the data. It is a combined wrapper of two counting semaphores and a FIFO; it has the most functions, but consumes the most resources and is the slowest.

Queue messages are all encapsulated with the doubly linked list head `struct RMP_List`. If a complex message must be passed, a message structure containing this structure can be defined, with `struct RMP_List` as its first element, so that the message structure can be found easily through the list head. For flexibility, the allocation and provision of the structure space are both done by the caller; the communication mechanism itself is not responsible for allocating or managing this space, which helps avoid unnecessary space waste and data copying. Although the queue is thread-safe, all messages are thread-local and queue-local:[1] two threads must not operate on one message at the same time, and the same message must not be inserted into two queues; once a message is enqueued, it must not be modified.

The message queue extension interface is not atomic; it is assembled from several atomic interfaces. Deleting a thread while it is using a message queue can leave the queue data structure incomplete. Deleting a message queue while it is in use and immediately creating a new queue with the same control block can cause an ABA problem. Therefore, before deleting a thread or a message queue, ensure that these situations do not exist, or perform a forced deletion in the following order: (1) delete all threads that use the message queue, (2) delete the message queue, and (3) re-create it if necessary. Note that in (1), some threads may have just allocated a message from the memory pool and have not yet linked it into the queue when they are deleted. Reclaiming the leaked message is then the application's responsibility; this can be done with a memory-block marking method, or by giving messages a dedicated memory pool and reinitialising that pool. In addition, a thread using a message queue can be suspended at any time, or its blocking on the queue can be released with `RMP_Sem_Abort`.

[1] Locking each message to prevent races costs too much performance and is not worth it.

### 3.1.2 Soft timer extension interface

For timed callbacks, RMP extends the software alarm interface. Through this interface, an application can register a callback function that is called after a period of time, implementing a timed-trigger function. To (1) improve the flexibility of the timer function and (2) prevent the timer function of a high-priority task from being disturbed by a low-priority task, RMP provides multiple timer managers: the priority at which each manager runs and the timing granularity it uses can be customised arbitrarily.

Therefore, the RMP soft-timer interface is divided into (1) timer-manager operations and (2) timer operations. To create a timer, the application must first (1) create a timer manager and maintain it periodically, then (2) initialise the required timer, and finally (3) register the timer with the corresponding timer manager. Once a timer times out, its callback hook is called; when the hook is called, the current manager `Amgr`, the current timer `Alrm`, and the overdue amount `Overdue` are passed as parameters to facilitate application design. Timers registered on the same manager always fire first by (1) timeout and then by (2) registration order: if one timer's timeout is earlier than another's, the earlier timer fires first; if two timers time out at the same time, the earlier-registered timer fires first.

Timer managers use `struct RMP_Amgr` and timers use `struct RMP_Alrm`. If data must be attached to these structures, a new structure containing the structure as its first element can be defined, so that the attached data can be found easily. For flexibility, allocation and provision of these structures are done by the caller; the alarm mechanism does not allocate or manage them, which avoids waste and copying. Although the manager is thread-safe, all timers are thread-local and manager-local:[2] two threads must not operate on one timer at the same time,[3] and the same timer must not be registered with two managers; once a timer is registered, it must not be reinitialised.[4]

The soft-timer extension interface is not atomic; it is assembled from several atomic interfaces. Deleting a thread while it is using the timer interface can leave the manager data structure incomplete. Deleting a manager while it is in use and immediately creating a new manager with the same control block can cause an ABA problem. Therefore, before deleting a thread or manager, ensure that these situations do not exist, or perform a forced deletion in the following order: (1) delete all threads that operate on the manager, (2) delete the manager, and (3) re-create it if necessary. Note that in (1), some threads may have just allocated a timer from the memory pool and have not yet linked it into the queue when they are deleted. Reclaiming the leaked timer is then the application's responsibility; this can be done with a memory-block marking method, or by giving timers a dedicated pool and reinitialising that pool. In addition, suspending a thread while it uses a timer is not recommended, because manager operations are protected by a semaphore[5] rather than the scheduler lock. If a thread is suspended while holding the semaphore, other threads using the same manager may be blocked indefinitely until the suspension is released.

## 3.2 First-in-first-out queue extension interface

All FIFO extension interfaces are listed below.

They may return the following values.

[2] Locking each timer to prevent races costs too much performance and is not worth it.

[3] Exception: two threads may operate on the same timer if both specify the same timer manager, because the manager's semaphore guarantees mutual exclusion.

[4] The interface detects this error in non-concurrent conditions and reports it, but does not guarantee detection in concurrent conditions.

[5] The semaphore is used here as a mutex.

Table 3-1. FIFO extension interface summary.

| Function | Purpose | Locked | ISR |
|---|---|---|---|
| `RMP_Fifo_Crt` | Create a FIFO. | ✓ | ✗ |
| `RMP_Fifo_Del` | Delete a FIFO. | ✓ | ✗ |
| `RMP_Fifo_Read` | Read from a FIFO. | ✓ | ✗ |
| `RMP_Fifo_Write` | Write to a FIFO. | ✓ | ✗ |
| `RMP_Fifo_Write_ISR` | Write to a FIFO from an interrupt. | ✗ | ✓ |
| `RMP_Fifo_Cnt` | Query the FIFO length. | ✓ | ✗ |

Table 3-2. Possible return values of the FIFO extension interfaces.

| Value | No. | Meaning |
|---|---|---|
| `RMP_ERR_OPER` | −6 | The operation failed for another reason. |
| `RMP_ERR_FIFO` | −9 | The FIFO control block caused the operation to fail. |

### 3.2.1 Create a FIFO

#### RMP_Fifo_Crt — Create a FIFO

```
rmp_ret_t RMP_Fifo_Crt(volatile struct RMP_Fifo* Fifo)
```

This operation creates a FIFO.

**Parameters**

**`volatile struct RMP_Fifo* Fifo`**
Pointer to an empty FIFO control block to be used for this FIFO.

**Returns**

**`rmp_ret_t`**
0 on success; `RMP_ERR_FIFO` if the control block is 0 (NULL) or already used.

### 3.2.2 Delete a FIFO

#### RMP_Fifo_Del — Delete a FIFO

```
rmp_ret_t RMP_Fifo_Del(volatile struct RMP_Fifo* Fifo)
```

This operation deletes a FIFO. Only an empty FIFO can be deleted.

**Parameters**

**`volatile struct RMP_Fifo* Fifo`**
Pointer to the FIFO control block to delete.

**Returns**

**`rmp_ret_t`**
0 on success. On failure: `RMP_ERR_FIFO` if the control block is 0 (NULL) or unused; `RMP_ERR_OPER` if the FIFO still contains elements and cannot be deleted. This check is performed even when `RMP_CHECK_ENABLE` is disabled.

### 3.2.3 Read from a FIFO

#### RMP_Fifo_Read

Read from a FIFO

```
rmp_ret_t RMP_Fifo_Read(volatile struct RMP_Fifo* Fifo, volatile struct RMP_List**
Node)
```

This operation tries to read one element from a FIFO. If the FIFO has no elements, the operation returns immediately and does not provide blocking.

**Parameters**

**`volatile struct RMP_Fifo* Fifo`**
Pointer to the FIFO control block to read.

**`volatile struct RMP_List** Node`**
Output parameter for the element read.

**Returns**

**`rmp_ret_t`**
0 on success. On failure: `RMP_ERR_FIFO` if the control block is 0 (NULL) or unused; `RMP_ERR_OPER` if the node pointer is 0 (NULL) or the FIFO temporarily has no elements. The last check is performed even when `RMP_CHECK_ENABLE` is disabled.

### 3.2.4 Write to a FIFO

#### RMP_Fifo_Write

Write to a FIFO

```
rmp_ret_t RMP_Fifo_Write(volatile struct RMP_Fifo* Fifo, volatile struct RMP_List*
Node)
```

This operation writes one element to a FIFO.

**Parameters**

**`volatile struct RMP_Fifo* Fifo`**
Pointer to the FIFO control block to write.

**`volatile struct RMP_List* Node`**
Pointer to the element to write.

**Returns**

**`rmp_ret_t`**
0 on success. On failure: `RMP_ERR_FIFO` if the control block is 0 (NULL) or unused; `RMP_ERR_OPER` if the write pointer is 0 (NULL).

### 3.2.5 Write to a FIFO from an interrupt

#### RMP_Fifo_Write_ISR

Write to a FIFO from an ISR

```
rmp_ret_t RMP_Fifo_Write_ISR(volatile struct RMP_Fifo* Fifo, volatile struct RMP_List*
Node)
```

This operation writes one element to a FIFO from an interrupt vector.

#### Parameters

**`volatile struct RMP_Fifo* Fifo`**
Pointer to the FIFO control block to write.

**`volatile struct RMP_List* Node`**
Pointer to the element to write.

#### Returns

**`rmp_ret_t`**
0 on success. On failure: `RMP_ERR_FIFO` if the control block is 0 (NULL) or unused; `RMP_ERR_OPER` if the write pointer is 0 (NULL).

### 3.2.6 Query the FIFO length

#### RMP_Fifo_Cnt — Query the FIFO length

```
rmp_ret_t RMP_Fifo_Cnt(volatile struct RMP_Fifo* Fifo)
```

This operation queries the current length of a FIFO.

#### Parameters

**`volatile struct RMP_Fifo* Fifo`**
Pointer to the FIFO control block whose length is to be queried.

#### Returns

**`rmp_ret_t`**
Current queue length on success; `RMP_ERR_FIFO` if the control block is 0 (NULL) or unused.

## 3.3 Message queue extension interface

All message queue extension interfaces are listed below.

Table 3-3. Message queue extension interface summary.

| Function | Purpose | Locked | ISR |
|---|---|---|---|
| `RMP_Msgq_Crt` | Create a message queue. | ✓ | ✗ |
| `RMP_Msgq_Del` | Delete a message queue. | ✓ | ✗ |
| `RMP_Msgq_Snd` | Send to a message queue. | ✓ | ✗ |
| `RMP_Msgq_Snd_ISR` | Send to a message queue from an interrupt. | ✗ | ✓ |
| `RMP_Msgq_Rcv` | Receive from a message queue. | ✗ | ✗ |
| `RMP_Msgq_Cnt` | Query the message queue length. | ✓ | ✗ |

They may return the following values.

Table 3-4. Possible return values of the message queue extension interfaces.

| Value | No. | Meaning |
|---|---|---|
| `RMP_ERR_OPER` | −6 | The operation failed for another reason. |
| `RMP_ERR_MSGQ` | −10 | The message queue control block caused the operation to fail. |

### 3.3.1 Create a message queue

#### RMP_Msgq_Crt

Create a message queue

```
rmp_ret_t RMP_Msgq_Crt(volatile struct RMP_Msgq* Queue)
```

This operation creates a message queue.

**Parameters**

**`volatile struct RMP_Msgq* Queue`**
Pointer to an empty message queue control block to be used for this message queue.

**Returns**

**`rmp_ret_t`**
0 on success; `RMP_ERR_MSGQ` if the control block is 0 (NULL) or already used.

### 3.3.2 Delete a message queue

#### RMP_Msgq_Del

Delete a message queue

```
rmp_ret_t RMP_Msgq_Del(volatile struct RMP_Msgq* Queue)
```

This operation deletes a message queue. Only an empty message queue can be deleted.

**Parameters**

**`volatile struct RMP_Msgq* Queue`**
Pointer to the message queue control block to delete.

**Returns**

**`rmp_ret_t`**
0 on success. On failure: `RMP_ERR_MSGQ` if the control block is 0 (NULL) or unused; `RMP_ERR_OPER` if the queue still contains elements and cannot be deleted. This check is performed even when `RMP_CHECK_ENABLE` is disabled.

### 3.3.3 Send to a message queue

#### RMP_Msgq_Snd

Send to a message queue

```
rmp_ret_t RMP_Msgq_Snd(volatile struct RMP_Msgq* Queue, volatile struct RMP_List* Node)
```

This operation sends one message to a message queue.

**Parameters**

**`volatile struct RMP_Msgq* Queue`**
Pointer to the message queue control block to send to.

**`volatile struct RMP_List* Node`**
Pointer to the message to send.

**Returns**

**`rmp_ret_t`**
0 on success. On failure: `RMP_ERR_MSGQ` if the control block is 0 (NULL) or unused; `RMP_ERR_OPER`

if the send pointer is 0 (NULL) or the current message count reaches `RMP_SEM_CNT_MAX`.

### 3.3.4 Send to a message queue from an interrupt

#### RMP_Msgq_Snd_ISR

Send to a message queue from an ISR

```
rmp_ret_t RMP_Msgq_Snd_ISR(volatile struct RMP_Msgq* Queue, volatile struct RMP_List*
Node)
```

This operation sends one message to a message queue from an interrupt. Unlike the ordinary version, it always returns immediately and does not block.

**Parameters**

**`volatile struct RMP_Msgq* Queue`**
Pointer to the message queue control block to send to.

**`volatile struct RMP_List* Node`**
Pointer to the message to send.

**Returns**

**`rmp_ret_t`**
0 on success. On failure: `RMP_ERR_MSGQ` if the control block is 0 (NULL) or unused; `RMP_ERR_OPER` if the send pointer is 0 (NULL) or the current message count reaches `RMP_SEM_CNT_MAX`.

### 3.3.5 Receive from a message queue

#### RMP_Msgq_Rcv

Receive from a message queue

```
rmp_ret_t RMP_Msgq_Rcv(volatile struct RMP_Msgq* Queue, volatile struct RMP_List**
Node, rmp_ptr_t Slice)
```

This operation receives one message from a message queue.

**Parameters**

**`volatile struct RMP_Msgq* Queue`**
Pointer to the message queue control block to receive from.

**`volatile struct RMP_List** Node`**
Output parameter for the received message.

**`rmp_ptr_t Slice`**
Number of time slices to wait when no message is available. If 0, return immediately if blocking is detected; if between 0 and `RMP_SLICE_MAX`, wait at most that many slices; if greater than or equal to `RMP_SLICE_MAX`, wait forever until a message is received or the queue is destroyed.

**Returns**

**`rmp_ret_t`**
0 on success. On failure: `RMP_ERR_MSGQ` if the control block is 0 (NULL) or unused; `RMP_ERR_OPER` if the receive pointer is 0 (NULL), blocking would occur with a zero timeout, or the wait timed out, was aborted, or the target queue was deleted. The last two checks are performed even when `RMP_CHECK_ENABLE` is disabled.

### 3.3.6 Query the message queue length

#### RMP_Msgq_Cnt

Query the message queue length

```
rmp_ret_t RMP_Msgq_Cnt(volatile struct RMP_Msgq* Queue)
```

This operation queries the current length of a message queue.

**Parameters**

**`volatile struct RMP_Msgq* Queue`**
Pointer to the message queue control block whose length is to be queried.

**Returns**

**`rmp_ret_t`**
Current queue length on success; `RMP_ERR_MSGQ` if the control block is 0 (NULL) or unused.

## 3.4 Blocking message queue extension interface

All blocking message queue extension interfaces are listed below.

Table 3-5. Blocking message queue extension interface summary.

| Function | Purpose | Locked | ISR |
|---|---|---|---|
| `RMP_Bmq_Crt` | Create a blocking message queue. | ✓ | ✗ |
| `RMP_Bmq_Del` | Delete a blocking message queue. | ✓ | ✗ |
| `RMP_Bmq_Snd` | Send to a blocking message queue. | ✗ | ✗ |
| `RMP_Bmq_Snd_ISR` | Send to a blocking message queue from an interrupt. | ✗ | ✓ |
| `RMP_Bmq_Rcv` | Receive from a blocking message queue. | ✗ | ✗ |
| `RMP_Bmq_Cnt` | Query the blocking message queue length. | ✓ | ✗ |

They may return the following values.

Table 3-6. Possible return values of the blocking message queue extension interfaces.

| Value | No. | Meaning |
|---|---|---|
| `RMP_ERR_OPER` | −6 | The operation failed for another reason. |
| `RMP_ERR_BMQ` | −11 | The blocking message queue control block caused the operation to fail. |

### 3.4.1 Create a blocking message queue

#### RMP_Bmq_Crt

Create a blocking message queue

```
rmp_ret_t RMP_Bmq_Crt(volatile struct RMP_Bmq* Queue, rmp_ptr_t Limit)
```

This operation creates a blocking message queue.

**Parameters**

**`volatile struct RMP_Bmq* Queue`**
Pointer to an empty blocking message queue control block.

**rmp_ptr_t Limit**
Maximum queue length: the maximum number of messages allowed in the queue at one time.

#### Returns

**rmp_ret_t**
0 on success. On failure: `RMP_ERR_BMQ` if the control block is 0 (NULL) or already used; `RMP_ERR_OPER` if the maximum length is 0 or reaches `RMP_SEM_CNT_MAX`.

### 3.4.2 Delete a blocking message queue

#### RMP_Bmq_Del

Delete a blocking message queue

```
rmp_ret_t RMP_Bmq_Del(volatile struct RMP_Bmq* Queue)
```

This operation deletes a blocking message queue. Only an empty blocking message queue can be deleted.

#### Parameters

**volatile struct RMP_Bmq* Queue**
Pointer to the blocking message queue control block to delete.

#### Returns

**rmp_ret_t**
0 on success. On failure: `RMP_ERR_BMQ` if the control block is 0 (NULL) or unused; `RMP_ERR_OPER` if the queue still contains elements and cannot be deleted. This check is performed even when `RMP_CHECK_ENABLE` is disabled.

### 3.4.3 Send to a blocking message queue

#### RMP_Bmq_Snd

Send to a blocking message queue

```
rmp_ret_t RMP_Bmq_Snd(volatile struct RMP_Bmq* Queue, volatile struct RMP_List* Node,
rmp_ptr_t Slice)
```

This operation sends one message to a blocking message queue.

#### Parameters

**volatile struct RMP_Bmq* Queue**
Pointer to the blocking message queue control block to send to.

**volatile struct RMP_List* Node**
Pointer to the message to send.

**rmp_ptr_t Slice**
Number of time slices to wait when the queue is full. If 0, return immediately if blocking would occur; if between 0 and `RMP_SLICE_MAX`, wait at most that many slices; if greater than or equal to `RMP_SLICE_MAX`, wait forever until the message is sent.

#### Returns

**rmp_ret_t**
0 on success. On failure: `RMP_ERR_BMQ` if the control block is 0 (NULL) or unused; `RMP_ERR_OPER` if the send pointer is 0 (NULL), the current message count reaches `RMP_SEM_CNT_MAX`, blocking would

occur with a zero timeout, or the wait timed out, was aborted, or the target queue was deleted. The last two checks are performed even when `RMP_CHECK_ENABLE` is disabled.

### 3.4.4 Send to a blocking message queue from an interrupt

#### RMP_Bmq_Snd_ISR

Send to a blocking queue from an ISR

```
rmp_ret_t RMP_Bmq_Snd_ISR(volatile struct RMP_Bmq* Queue, volatile struct RMP_List*
Node)
```

This operation sends one element to a blocking message queue from an interrupt. Unlike the ordinary version, it always returns immediately, does not block, and does not accept a timeout parameter.

##### Parameters

**`volatile struct RMP_Bmq* Queue`**
Pointer to the blocking message queue control block to send to.

**`volatile struct RMP_List* Node`**
Pointer to the message to send.

##### Returns

**`rmp_ret_t`**
0 on success. On failure: `RMP_ERR_BMQ` if the control block is 0 (NULL) or unused; `RMP_ERR_OPER` if the send pointer is 0 (NULL), the current message count reaches `RMP_SEM_CNT_MAX`, or the message count reaches the maximum allowed at creation. The last check is performed even when `RMP_CHECK_ENABLE` is disabled.

### 3.4.5 Receive from a blocking message queue

#### RMP_Bmq_Rcv

Receive from a blocking message queue

```
rmp_ret_t RMP_Bmq_Rcv(volatile struct RMP_Bmq* Queue, volatile struct RMP_List** Node,
rmp_ptr_t Slice)
```

This operation receives one message from a blocking message queue.

##### Parameters

**`volatile struct RMP_Bmq* Queue`**
Pointer to the blocking message queue control block to receive from.

**`volatile struct RMP_List** Node`**
Output parameter for the received message.

**`rmp_ptr_t Slice`**
Number of time slices to wait when no message is available. If 0, return immediately if blocking would occur; if between 0 and `RMP_SLICE_MAX`, wait at most that many slices; if greater than or equal to `RMP_SLICE_MAX`, wait forever until a message is received or the queue is destroyed.

##### Returns

**`rmp_ret_t`**
0 on success. On failure: `RMP_ERR_BMQ` if the control block is 0 (NULL) or unused; `RMP_ERR_OPER` if

the receive pointer is 0 (NULL), blocking would occur with a zero timeout, or the wait timed out, was aborted, or the queue was deleted. The last two checks are performed even when `RMP_CHECK_ENABLE` is disabled.

### 3.4.6 Query the blocking message queue length

#### RMP_Bmq_Cnt

Query the blocking queue length

```
rmp_ret_t RMP_Bmq_Cnt(volatile struct RMP_Bmq* Queue)
```

This operation queries the current length of a blocking message queue.

**Parameters**

**`volatile struct RMP_Bmq* Queue`**
Pointer to the blocking message queue control block whose length is to be queried.

**Returns**

**`rmp_ret_t`**
Current queue length on success; `RMP_ERR_BMQ` if the control block is 0 (NULL) or unused.

## 3.5 Soft timer extension interface

All soft timer extension interfaces are listed below.

Table 3-7. Soft timer extension interface summary.

| Function | Purpose | Locked | ISR |
|---|---|---|---|
| `RMP_Amgr_Crt` | Create an alarm manager. | ✓ | ✗ |
| `RMP_Amgr_Del` | Delete an alarm manager. | ✓ | ✗ |
| `RMP_Amgr_Pop` | Dismantle an alarm manager. | ✓ | ✗ |
| `RMP_Amgr_Proc` | Maintain an alarm manager. | ✗ | ✗ |
| `RMP_Amgr_Cnt` | Query the number of alarms. | ✓ | ✗ |
| `RMP_Alrm_Init` | Initialise an alarm. | ✓ | ✗ |
| `RMP_Alrm_Set` | Register an alarm. | ✗ | ✗ |
| `RMP_Alrm_Clr` | Cancel an alarm. | ✗ | ✗ |
| `RMP_Alrm_Trg` | Trigger an alarm. | ✗ | ✗ |

They may return the following values.

Table 3-8. Possible return values of the soft timer extension interfaces.

| Value | No. | Meaning |
|---|---|---|
| `RMP_ERR_OPER` | −6 | The operation failed for another reason. |
| `RMP_ERR_AMGR` | −12 | The alarm manager control block caused the operation to fail. |
| `RMP_ERR_ALRM` | −13 | The alarm control block caused the operation to fail. |

### 3.5.1 Create an alarm manager

#### RMP_Amgr_Crt

Create an alarm manager

```
rmp_ret_t RMP_Amgr_Crt(volatile struct RMP_Amgr* Amgr)
```

This operation creates an alarm manager.

**Parameters**

**`volatile struct RMP_Amgr* Amgr`**
Pointer to an empty alarm manager control block.

**Returns**

**`rmp_ret_t`**
0 on success; `RMP_ERR_AMGR` if the control block is 0 (NULL) or already used.

### 3.5.2 Delete an alarm manager

#### RMP_Amgr_Del

Delete an alarm manager

```
rmp_ret_t RMP_Amgr_Del(volatile struct RMP_Amgr* Amgr)
```

This operation deletes an alarm manager. Only an empty alarm manager can be deleted. If the manager is to be deleted, either wait for (1) its alarms to expire naturally or (2) dismantle it manually with the function described in the next section.

**Parameters**

**`volatile struct RMP_Amgr* Amgr`**
Pointer to the alarm manager control block to delete.

**Returns**

**`rmp_ret_t`**
0 on success. On failure: `RMP_ERR_AMGR` if the control block is 0 (NULL) or unused; `RMP_ERR_OPER` if the manager still contains alarms and cannot be deleted. This check is performed even when `RMP_CHECK_ENABLE` is disabled.

### 3.5.3 Dismantle an alarm manager

#### RMP_Amgr_Pop

Dismantle an alarm manager

```
rmp_ret_t RMP_Amgr_Pop(volatile struct RMP_Amgr* Amgr, volatile struct RMP_Alrm** Alrm)
```

This operation forcibly pops one alarm from an alarm manager. When an alarm manager must be stopped forcibly and, after all threads that may operate on it have been terminated, its data structure may be damaged, this function can be used to dismantle the manager step by step so that it no longer contains any alarms and can be deleted. If the alarms were allocated from a memory pool, they can be returned to the pool one by one. Before dismantling, all manager operations must stop; once dismantling starts, only dismantling or deletion of the manager is allowed.

**Parameters**

**volatile struct RMP_Amgr* Amgr**
Pointer to the alarm manager control block to dismantle.

**volatile struct RMP_Alrm** Alrm**
Output parameter for the dismantled alarm.

#### Returns

**rmp_ret_t**
0 on success. On failure: `RMP_ERR_AMGR` if the control block is 0 (NULL) or unused; `RMP_ERR_OPER` if the output pointer is 0 (NULL) or the manager has no alarms. The last check is performed even when `RMP_CHECK_ENABLE` is disabled.

### 3.5.4 Maintain an alarm manager

#### RMP_Amgr_Proc

Maintain an alarm manager

```
rmp_ret_t RMP_Amgr_Proc(volatile struct RMP_Amgr* Amgr, rmp_ptr_t Tick)
```

This operation adds to the manager's time count and processes any alarms that have expired; it is the core interface of the alarm manager. It must be called periodically to process alarms registered on the manager; a dedicated thread can be arranged to call it periodically, and the call period is the manager's timing granularity.

#### Parameters

**volatile struct RMP_Amgr* Amgr**
Pointer to the alarm manager control block to maintain.

**rmp_ptr_t Tick**
Time count to add to the manager. The number is not limited by `RMP_SLICE_MAX`, but it must not be 0 or exceed $2^{N-1}-1$ (`RMP_MASK_INTMAX`), where $N$ is `RMP_WORD_ORDER`. Adding more than one count at once may cause an alarm to expire late; the callback's overdue amount will then be positive.

#### Returns

**rmp_ret_t**
Current number of alarms on the manager on success. On failure: `RMP_ERR_AMGR` if the control block is 0 (NULL) or unused; `RMP_ERR_OPER` if `Tick` is 0 or exceeds `RMP_MASK_INTMAX`, or locking fails because the manager was deleted or cancelled. The last check is performed even when `RMP_CHECK_ENABLE` is disabled.

### 3.5.5 Query the number of alarms

#### RMP_Amgr_Cnt

Query the number of alarms

```
rmp_ret_t RMP_Amgr_Cnt(volatile struct RMP_Amgr* Amgr)
```

This operation queries the current number of alarms on an alarm manager.

#### Parameters

**volatile struct RMP_Amgr* Amgr**
Pointer to the alarm manager control block whose alarm count is to be queried.

#### Returns

**rmp_ret_t**
Current number of alarms on success; `RMP_ERR_AMGR` if the control block is 0 (NULL) or unused.

### 3.5.6 Initialise an alarm

#### RMP_Alrm_Init

Initialise an alarm

```
rmp_ret_t RMP_Alrm_Init(volatile struct RMP_Alrm* Alrm, rmp_ptr_t Delay, rmp_ptr_t
Mode,
void (*Hook)(volatile struct RMP_Amgr*, volatile struct RMP_Alrm*, rmp_cnt_t))
```

This operation initialises an alarm's data structure according to the parameters given by the user, so that it can be registered with an alarm manager next.

##### Parameters

**volatile struct RMP_Alrm* Alrm**
Pointer to the alarm control block to initialise.

**rmp_ptr_t Delay**
Delay of the alarm. The number is not limited by `RMP_SLICE_MAX`, but it must not be 0 or exceed $2^{N-1} - 1$ (`RMP_MASK_INTMAX`). Its unit is the manager's timing granularity.

**rmp_ptr_t Mode**
Working mode of the alarm, either `RMP_ALRM_AUTORLD` (auto-reload) or `RMP_ALRM_ONESHOT`.

**void (*Hook)(volatile struct RMP_Amgr*, volatile struct RMP_Alrm*, rmp_cnt_t) Hook**
Callback hook of the alarm. Its prototype is `void Hook(volatile struct RMP_Amgr* Amgr, volatile struct RMP_Alrm* Alrm, rmp_cnt_t Overdue)`.

##### Returns

**rmp_ret_t**
0 on success. On failure: `RMP_ERR_ALRM` if the control block is 0 (NULL) or already registered; `RMP_ERR_OPER` if `Delay` is 0 or exceeds `RMP_MASK_INTMAX`, `Mode` is neither of the two allowed values, or `Hook` is 0 (NULL).

#### Hook

Alarm callback hook

```
void Hook(volatile struct RMP_Amgr* Amgr, volatile struct RMP_Alrm* Alrm, rmp_cnt_t
Overdue)
```

##### Parameters

**volatile struct RMP_Amgr* Amgr**
Pointer to the alarm manager to which this alarm belongs.

**volatile struct RMP_Alrm* Alrm**
Pointer to this alarm's control block.

**rmp_cnt_t Overdue**
How overdue the alarm is. Positive if the alarm expired late because `RMP_Amgr_Proc` advanced the count by more than one tick; negative if it was triggered early by `RMP_Alrm_Trg`; 0 if it expired normally.

##### Returns

**None**
No return value.

### 3.5.7 Register an alarm

#### RMP_Alrm_Set

Register an alarm

```
rmp_ret_t RMP_Alrm_Set(volatile struct RMP_Amgr* Amgr, volatile struct RMP_Alrm* Alrm)
```

This operation registers the specified alarm with the specified alarm manager. If the alarm is already registered with the same manager, its delay is reset.

Parameters

**`volatile struct RMP_Amgr* Amgr`**
Pointer to the alarm manager control block to register with.

**`volatile struct RMP_Alrm* Alrm`**
Pointer to the alarm control block to register.

Returns

**`rmp_ret_t`**
0 on success. On failure: `RMP_ERR_AMGR` if the manager control block is 0 (NULL) or unused; `RMP_ERR_ALRM` if the alarm control block is 0 (NULL) or registered with another manager (the last check is performed even when `RMP_CHECK_ENABLE` is disabled); `RMP_ERR_OPER` if locking fails because the manager was deleted or cancelled (also checked even when checking is disabled).

### 3.5.8 Cancel an alarm

#### RMP_Alrm_Clr

Cancel an alarm

```
rmp_ret_t RMP_Alrm_Clr(volatile struct RMP_Amgr* Amgr, volatile struct RMP_Alrm* Alrm)
```

This operation cancels the specified alarm. The cancelled alarm is immediately unregistered from the manager.

Parameters

**`volatile struct RMP_Amgr* Amgr`**
Pointer to the alarm manager control block that owns the alarm.

**`volatile struct RMP_Alrm* Alrm`**
Pointer to the alarm control block to cancel.

Returns

**`rmp_ret_t`**
0 on success. On failure: `RMP_ERR_AMGR` if the manager control block is 0 (NULL) or unused; `RMP_ERR_ALRM` if the alarm control block is 0 (NULL) or not registered with this manager (the last check is performed even when `RMP_CHECK_ENABLE` is disabled); `RMP_ERR_OPER` if locking fails because the manager was deleted or cancelled (also checked even when checking is disabled).

### 3.5.9 Trigger an alarm

#### RMP_Alrm_Trg

Trigger an alarm

```
rmp_ret_t RMP_Alrm_Trg(volatile struct RMP_Amgr* Amgr, volatile struct RMP_Alrm* Alrm)
```

This operation triggers the specified alarm early, as if its delay had expired. The alarm's callback hook is called, and its overdue amount is negative. Afterwards, an auto-reload alarm starts counting again immediately, while a one-shot alarm is unregistered from its manager.

##### Parameters

**`volatile struct RMP_Amgr* Amgr`**
Pointer to the alarm manager control block that owns the alarm.

**`volatile struct RMP_Alrm* Alrm`**
Pointer to the alarm control block to trigger early.

##### Returns

**`rmp_ret_t`**
0 on success. On failure: `RMP_ERR_AMGR` if the manager control block is 0 (NULL) or unused; `RMP_ERR_ALRM` if the alarm control block is 0 (NULL) or not registered with this manager (the last check is performed even when `RMP_CHECK_ENABLE` is disabled); `RMP_ERR_OPER` if locking fails because the manager was deleted or cancelled (also checked even when checking is disabled).

## 3.6 Chapter bibliography

None.

# 4. Lightweight Graphics Interface

## 4.1 Introduction to embedded graphical interfaces

An embedded graphical user interface (GUI) is a graphics solution dedicated to embedded systems. Compared with a traditional computer GUI, an embedded GUI has the characteristics of simplicity, diversity, planarity, and restriction. Grasping these four characteristics is the key to a good embedded GUI design.

### *4.1.1 Simplicity*

Generally, the elements of an embedded GUI are relatively simple: usually a few buttons or a few display controls, unlike a traditional GUI with scroll bars, tree lists, and many other elements. Moreover, once the UI is designed, it rarely changes during the product's life.

### *4.1.2 Diversity*

After years of evolution, traditional GUI design has reached a certain consensus, forming a large number of desktop environments represented by Windows, MacOS, Gnome, and Unity. They all emphasise optimising the mouse and keyboard experience, use almost the same interaction logic, and mostly have standardised controls. Embedded GUIs are the opposite: standard mouse and keyboard input is rarely used; custom keypads and touch screens are more common, and there are many non-standard controls and interfaces. The appearance and logic of these controls differ, and require a great deal of optimisation and adaptation for the specific application.

### *4.1.3 Planarity*

The traditional GUI technology stack places great emphasis on complex window clipping, so that windows have a vertical stacking relationship; complex algorithms calculate the display region of each window and draw it. This is necessary in extremely complex multi-window, multi-task environments. In an embedded GUI, the display area is small and the input means are limited, so vertical spatial relationships between windows are rarely used; flat and humane GUI design is emphasised instead. This can also be seen in the evolution of mobile operating systems.

### *4.1.4 Restriction*

Traditional GUI implementations often depend on powerful CPU and GPU support and large amounts of memory for complex graphics operations. In an embedded environment, it is often difficult even to obtain enough frame-buffer memory, let alone use dedicated hardware for graphics. A GUI for an embedded environment therefore needs special optimisation.

## 4.2 Enabling and configuring the graphics interface

RMP supports a GUI by providing a series of drawing functions. Its basic drawing functions include point and line drawing, geometric-shape drawing, and bitmap anti-aliasing rendering; the more complex func-

tions include drawing various controls. These drawing functions do not record the state of GUI objects and do not use object-oriented design, so they consume no extra RAM and only a certain amount of ROM.

The drawing functions provided by RMP are divided into basic drawing functions, anti-aliased drawing functions, and control drawing functions. To use them, the corresponding switches must be enabled in the configuration header, and some additional low-level macros must be provided. These macros are listed in Table 4-1.

Table 4-1. Macros required to use the built-in embedded graphics interface.

| Macro | Meaning |
|---|---|
| `RMP_GUI_ENABLE` | Master switch for the GUI. Define as 1 to enable GUI functions. Example: `#define RMP_GUI_ENABLE 1U`. |
| `RMP_POINT(X,Y,COLOR)` | Macro for the point-drawing function. Once the GUI is enabled, this macro should be defined as the name of a point-drawing function with the prototype `void foo(rmp_cnt_t X, rmp_cnt_t Y, rmp_ptr_t Color)`. Its purpose is to draw one point on the screen. Example: `#define RMP_POINT RMP_Point`. |
| `RMP_GUI_ANTIALIAS_ENABLE` | Master switch for anti-aliased drawing. Define as 1 to enable it. Example: `#define RMP_GUI_ANTIALIAS_ENABLE 1U`. |
| `RMP_COLOR_25P(C1,C2)` | Colour-mixing macros. If anti-aliased drawing is enabled, these macros must be defined. Each takes two colours and returns a colour. `RMP_COLOR_25P` mixes 25% of C1 with 75% of C2; `RMP_COLOR_50P` mixes them equally; `RMP_COLOR_75P` mixes 75% of C1 with 25% of C2. |
| `RMP_COLOR_50P(C1,C2)` | See above. |
| `RMP_COLOR_75P(C1,C2)` | See above. |
| `RMP_GUI_WIDGET_ENABLE` | Master switch for control drawing. Define as 1 to enable it. Example: `#define RMP_GUI_WIDGET_ENABLE 1U`. |
| `RMP_COLOR_WHITE` | Built-in control colour macros. If control drawing is enabled, these macros must be defined. White, recommended `0xFFFFFF`. |
| `RMP_COLOR_LGREY` | Light grey, recommended `0xB8B8B8`. |
| `RMP_COLOR_GREY` | Grey, recommended `0xE0E0E0`. |
| `RMP_COLOR_DGREY` | Dark grey, recommended `0xA0A0A0`. |
| `RMP_COLOR_DARK` | Dark, recommended `0x787C78`. |
| `RMP_COLOR_DDARK` | Very dark, recommended `0x686868`. |
| `RMP_COLOR_BLACK` | Black, recommended `0x000000`. |

The drawing functions provided by RMP do not check whether the X and Y coordinates of the drawing range exceed the screen boundary. This responsibility belongs to the application developer.

## 4.3 Basic drawing functions

The basic drawing functions require `RMP_GUI_ENABLE` and a definition of `RMP_POINT`.

### *4.3.1 Draw a line*

#### RMP_Line

Draw a line

```
void RMP_Line(rmp_cnt_t Begin_X, rmp_cnt_t Begin_Y, rmp_cnt_t End_X, rmp_cnt_t End_Y,
rmp_ptr_t Color)
```

This function draws a line of width 1 on the screen. Rasterisation uses the Bresenham algorithm.

**Parameters**

**rmp_cnt_t Begin_X**
X coordinate of the line's start point.

**rmp_cnt_t Begin_Y**
Y coordinate of the line's start point.

**rmp_cnt_t End_X**
X coordinate of the line's end point.

**rmp_cnt_t End_Y**
Y coordinate of the line's end point.

**rmp_ptr_t Color**
Colour of the line.

#### Returns

**None**
No return value.

### *4.3.2 Draw a dotted line*

#### RMP_Dot_Line

Draw a dotted line

```
void RMP_Dot_Line(rmp_cnt_t Begin_X, rmp_cnt_t Begin_Y, rmp_cnt_t End_X, rmp_cnt_t
End_Y,
rmp_ptr_t Dot, rmp_ptr_t Space)
```

This function draws a dotted line of width 1 on the screen. Rasterisation uses the Bresenham algorithm.

#### Parameters

**rmp_cnt_t Begin_X**
X coordinate of the dotted line's start point.

**rmp_cnt_t Begin_Y**
Y coordinate of the dotted line's start point.

**rmp_cnt_t End_X**
X coordinate of the dotted line's end point.

**rmp_cnt_t End_Y**
Y coordinate of the dotted line's end point.

**rmp_ptr_t Dot**
Colour of the dots.

**rmp_ptr_t Space**
Colour of the gaps. If `RMP_TRANS` is entered, the gaps are not drawn.

#### Returns

**None**
No return value.

### 4.3.3 Draw a rectangle

#### RMP_Rectangle

Draw a rectangle

```
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)
```

This function draws a rectangle on the screen. The rectangle can have a border colour different from the fill colour, and it can be filled or not filled.

##### Parameters

**`rmp_cnt_t Coord_X`**
X coordinate of the rectangle's top-left corner.

**`rmp_cnt_t Coord_Y`**
Y coordinate of the rectangle's top-left corner.

**`rmp_cnt_t Length`**
Length of the rectangle.

**`rmp_cnt_t Width`**
Width of the rectangle.

**`rmp_ptr_t Border`**
Border colour of the rectangle.

**`rmp_ptr_t Fill`**
Fill colour of the rectangle. If `RMP_TRANS` is entered, the rectangle is not filled.

##### Returns

**`None`**
No return value.

### 4.3.4 Draw a rounded rectangle

#### RMP_Round_Rect

Draw a rounded rectangle

```
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)
```

This function draws a rounded rectangle on the screen. The outline and fill colour of the rounded rectangle are the same, and it is always completely filled.

##### Parameters

**`rmp_cnt_t Coord_X`**
X coordinate of the rectangle's top-left corner.

**`rmp_cnt_t Coord_Y`**
Y coordinate of the rectangle's top-left corner.

**`rmp_cnt_t Length`**
Length of the rectangle.

**`rmp_cnt_t Width`**
Width of the rectangle.

**rmp_cnt_t Round**
Corner radius of the rounded rectangle.

**rmp_ptr_t Color**
Colour of the rounded rectangle.

#### Returns

**None**
No return value.

### 4.3.5 Draw a circle

#### RMP_Circle

Draw a circle

```
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)
```

This function draws a circle on the screen. The circle can have a border colour different from the fill colour, and it can be filled or not filled.

#### Parameters

**rmp_cnt_t Center_X**
X coordinate of the circle's centre.

**rmp_cnt_t Center_Y**
Y coordinate of the circle's centre.

**rmp_cnt_t Radius**
Radius of the circle.

**rmp_ptr_t Border**
Border colour of the circle.

**rmp_ptr_t Fill**
Fill colour of the circle. If `RMP_TRANS` is entered, the circle is not filled.

#### Returns

**None**
No return value.

### 4.3.6 Draw a monochrome bitmap

#### RMP_Matrix

Draw a monochrome bitmap

```
void RMP_Matrix(rmp_cnt_t Coord_X, rmp_cnt_t Coord_Y, const rmp_u8_t* Matrix, rmp_ptr_t
Bit_Order,
rmp_cnt_t Length, rmp_cnt_t Width, rmp_ptr_t Color)
```

This function draws a monochrome bitmap on the screen. The bitmap length must be a multiple of 8, and the bitmap must be stored row-major, scanned from left to right and from high to low. High-bit-first or low-bit-first mode can be selected when drawing; in the former, the highest bit of a byte has the smallest X value among the pixels represented by that byte, and in the latter the opposite is true.[a]

#### Parameters

**rmp_cnt_t Coord_X**
X coordinate of the bitmap's top-left corner.

**rmp_cnt_t Coord_Y**
Y coordinate of the bitmap's top-left corner.

**const rmp_u8_t* Matrix**
Pointer to the monochrome bitmap data.

**rmp_ptr_t Bit_Order**
Bit order within a byte. `RMP_MAT_BIG` is high-bit-first; `RMP_MAT_SMALL` is low-bit-first.

**rmp_cnt_t Length**
Bitmap length; must be a multiple of 8, otherwise drawing does not start.

**rmp_cnt_t Width**
Bitmap width.

**rmp_ptr_t Color**
Colour used to draw the bitmap.

### Returns

**None**
No return value.

[a] This is similar to the concept of endianness, but here it is the pixel order of the bits within one byte.

## 4.4 Anti-aliased drawing functions

The anti-aliased function requires `RMP_GUI_ANTIALIAS_ENABLE` and the three colour-mixing macros. It draws a monochrome bitmap in 4× FXAA anti-aliased mode. The bitmap length must be a multiple of 8, and the bitmap is stored row-major, scanned from left to right and from high to low. The bit order inside a byte can be most-significant-bit first or least-significant-bit first.[1]

### RMP_Matrix_AA

Draw an anti-aliased monochrome bitmap

```
void RMP_Matrix_AA(rmp_cnt_t Coord_X, rmp_cnt_t Coord_Y, const rmp_u8_t* Matrix,
rmp_ptr_t Bit_Order, rmp_cnt_t Length, rmp_cnt_t Width,
rmp_ptr_t Color, rmp_ptr_t Back)
```

This function draws a monochrome bitmap on the screen in a 4× FXAA anti-aliased mode. The bitmap length must be a multiple of 8, and the bitmap must be stored row-major, scanned from left to right and from high to low. High-bit-first or low-bit-first mode can be selected. The background colour of the layer containing the bitmap must be passed for anti-aliasing. The three colour-mixing macros must also be defined.

### Parameters

**rmp_cnt_t Coord_X**
X coordinate of the bitmap's top-left corner.

**rmp_cnt_t Coord_Y**
Y coordinate of the bitmap's top-left corner.

**const rmp_u8_t* Matrix**
Pointer to the monochrome bitmap data.

[1] This is similar to endianness, but here it is the pixel order within one byte.

**rmp_ptr_t Bit_Order**
Bit order within a byte; `RMP_MAT_BIG` or `RMP_MAT_SMALL`.

**rmp_cnt_t Length**
Bitmap length; must be a multiple of 8, otherwise drawing does not start.

**rmp_cnt_t Width**
Bitmap width.

**rmp_ptr_t Color**
Colour used to draw the bitmap.

**rmp_ptr_t Back**
Background colour of the layer containing the bitmap.

#### Returns

**None**
No return value.

## 4.5 Widget drawing functions

The widget functions require `RMP_GUI_WIDGET_ENABLE` and the seven widget colour macros.

### *4.5.1 Draw a cursor*

#### RMP_Cursor — Draw a cursor

```
void RMP_Cursor(rmp_cnt_t Coord_X, rmp_cnt_t Coord_Y, rmp_ptr_t Style)
```

This function can draw various 16×16 cursors on the screen.

#### Parameters

**rmp_cnt_t Coord_X**
X coordinate of the cursor's top-left corner.

**rmp_cnt_t Coord_Y**
Y coordinate of the cursor's top-left corner.

**rmp_ptr_t Style**
Cursor style, one of `RMP_CUR_NORM`, `RMP_CUR_BUSY`, `RMP_CUR_QUESTION`, `RMP_CUR_HAND`, `RMP_CUR_TEXT`, `RMP_CUR_STOP`, `RMP_CUR_MOVE`, `RMP_CUR_LR`, `RMP_CUR_UD`, `RMP_CUR_ULBR`, `RMP_CUR_URBL`, or `RMP_CUR_CROSS`.

#### Returns

**None**
No return value.

### *4.5.2 Draw a checkbox*

#### RMP_Checkbox — Draw a checkbox

```
void RMP_Checkbox(rmp_cnt_t Coord_X, rmp_cnt_t Coord_Y, rmp_cnt_t Length, rmp_ptr_t Status)
```

This function draws a checkbox of variable size on the screen.

#### Parameters

**`rmp_cnt_t Coord_X`**
X coordinate of the checkbox's top-left corner.

**`rmp_cnt_t Coord_Y`**
Y coordinate of the checkbox's top-left corner.

**`rmp_cnt_t Length`**
Length of the checkbox.[a]

**`rmp_ptr_t Status`**
Current state of the checkbox. If `RMP_CBOX_CHECK` is entered, the checkbox is checked; 0 means unchecked.

#### Returns

**`None`**
No return value.

[a] This is also the width.

### 4.5.3 Check a checkbox

## RMP_Checkbox_Set

Check a checkbox

```
void RMP_Checkbox_Set(rmp_cnt_t Coord_X, rmp_cnt_t Coord_Y, rmp_cnt_t Length)
```

This function checks an already drawn checkbox.

#### Parameters

**`rmp_cnt_t Coord_X`**
X coordinate of the checkbox's top-left corner.

**`rmp_cnt_t Coord_Y`**
Y coordinate of the checkbox's top-left corner.

**`rmp_cnt_t Length`**
Length of the checkbox.[a]

#### Returns

**`None`**
No return value.

[a] This is also the width.

### 4.5.4 Clear a checkbox

## RMP_Checkbox_Clr

Clear a checkbox

```
void RMP_Checkbox_Clr(rmp_cnt_t Coord_X, rmp_cnt_t Coord_Y, rmp_cnt_t Length)
```

This function clears an already drawn checkbox.

#### Parameters

**rmp_cnt_t Coord_X**
X coordinate of the checkbox's top-left corner.

**rmp_cnt_t Coord_Y**
Y coordinate of the checkbox's top-left corner.

**rmp_cnt_t Length**
Length of the checkbox.[a]

### Returns

**None**
No return value.

[a] This is also the width.

## 4.5.5 Draw a command button

### RMP_Cmdbtn

Draw a command button

```
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)
```

This function draws a command button of variable size.

### Parameters

**rmp_cnt_t Coord_X**
X coordinate of the button's top-left corner.

**rmp_cnt_t Coord_Y**
Y coordinate of the button's top-left corner.

**rmp_cnt_t Length**
Length of the button.

**rmp_cnt_t Width**
Width of the button.

**rmp_ptr_t Status**
Current state of the button. If `RMP_CBTN_DOWN` is entered, the button is pressed; 0 means released.

### Returns

**None**
No return value.

## 4.5.6 Press a command button

### RMP_Cmdbtn_Down

Press a command button

```
void RMP_Cmdbtn_Down(rmp_cnt_t Coord_X, rmp_cnt_t Coord_Y, rmp_cnt_t Length, rmp_cnt_t Width)
```

This function presses an already drawn button.

### Parameters

**rmp_cnt_t Coord_X**
X coordinate of the button's top-left corner.

**rmp_cnt_t Coord_Y**
Y coordinate of the button's top-left corner.

**rmp_cnt_t Length**
Length of the button.

**rmp_cnt_t Width**
Width of the button.

#### Returns

**None**
No return value.

### 4.5.7 Release a command button

#### RMP_Cmdbtn_Up

Release a command button

```
void RMP_Cmdbtn_Up(rmp_cnt_t Coord_X, rmp_cnt_t Coord_Y, rmp_cnt_t Length, rmp_cnt_t Width)
```

This function releases an already drawn button.

#### Parameters

**rmp_cnt_t Coord_X**
X coordinate of the button's top-left corner.

**rmp_cnt_t Coord_Y**
Y coordinate of the button's top-left corner.

**rmp_cnt_t Length**
Length of the button.

**rmp_cnt_t Width**
Width of the button.

#### Returns

**None**
No return value.

### 4.5.8 Draw a text-edit box

#### RMP_Lineedit

Draw a text-edit box

```
void RMP_Lineedit(rmp_cnt_t Coord_X, rmp_cnt_t Coord_Y, rmp_cnt_t Length, rmp_cnt_t Width)
```

This function draws a text-edit box of variable size.

#### Parameters

**rmp_cnt_t Coord_X**
X coordinate of the text-edit box's top-left corner.

**rmp_cnt_t Coord_Y**
Y coordinate of the text-edit box's top-left corner.

**rmp_cnt_t Length**
Length of the text-edit box.

**rmp_cnt_t Width**
Width of the text-edit box.

#### Returns

**None**
No return value.

### 4.5.9 Clear part of a text-edit box

#### RMP_Lineedit_Clr

Clear part of a text-edit box

```
void RMP_Lineedit_Clr(rmp_cnt_t Clr_X, rmp_cnt_t Clr_Len, rmp_cnt_t Coord_Y, rmp_cnt_t Width)
```

This function clears part of a text-edit box, implementing text deletion in the box.

#### Parameters

**rmp_cnt_t Clr_X**
Starting X coordinate of the part to clear. This is a relative coordinate value.

**rmp_cnt_t Clr_Len**
Length to clear.

**rmp_cnt_t Coord_Y**
Y coordinate of the text-edit box's top-left corner.

**rmp_cnt_t Width**
Width of the text-edit box.

#### Returns

**None**
No return value.

### 4.5.10 Draw a radio button

#### RMP_Radiobtn

Draw a radio button

```
void RMP_Radiobtn(rmp_cnt_t Coord_X, rmp_cnt_t Coord_Y, rmp_cnt_t Length, rmp_ptr_t Status)
```

This function draws a radio button of variable size.

#### Parameters

**rmp_cnt_t Coord_X**
X coordinate of the radio button's top-left corner.

**rmp_cnt_t Coord_Y**
Y coordinate of the radio button's top-left corner.

**rmp_cnt_t Length**
Length of the radio button.[a]

**rmp_ptr_t Status**
Current state of the radio button. If `RMP_RBTN_SEL` is entered, the radio button is selected; 0 means not selected.

### Returns

**None**
No return value.

[a] This is also the width.

## 4.5.11 Select a radio button

### RMP_Radiobtn_Set

Select a radio button

```
void RMP_Radiobtn_Set(rmp_cnt_t Coord_X, rmp_cnt_t Coord_Y, rmp_cnt_t Length)
```

This function selects an already drawn radio button.

### Parameters

**rmp_cnt_t Coord_X**
X coordinate of the radio button's top-left corner.

**rmp_cnt_t Coord_Y**
Y coordinate of the radio button's top-left corner.

**rmp_cnt_t Length**
Length of the radio button.[a]

### Returns

**None**
No return value.

[a] This is also the width.

## 4.5.12 Clear a radio button

### RMP_Radiobtn_Clr

Clear a radio button

```
void RMP_Radiobtn_Clr(rmp_cnt_t Coord_X, rmp_cnt_t Coord_Y, rmp_cnt_t Length)
```

This function clears an already drawn radio button.

### Parameters

**rmp_cnt_t Coord_X**
X coordinate of the radio button's top-left corner.

**rmp_cnt_t Coord_Y**
Y coordinate of the radio button's top-left corner.

**rmp_cnt_t Length**
Length of the radio button.[a]

### Returns

**None**
No return value.

[a] This is also the width.

### 4.5.13 Draw a progress bar

#### RMP_Progbar

Draw a progress bar

```
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)
```

This function draws a progress bar.

Parameters

**`rmp_cnt_t Coord_X`**
X coordinate of the progress bar's top-left corner.

**`rmp_cnt_t Coord_Y`**
Y coordinate of the progress bar's top-left corner.

**`rmp_cnt_t Length`**
Length of the progress bar.

**`rmp_cnt_t Width`**
Width of the progress bar.

**`rmp_cnt_t Style`**
Style of the progress bar, one of `RMP_PBAR_L2R` (grows left to right), `RMP_PBAR_D2U` (grows down to up), `RMP_PBAR_R2L` (grows right to left), or `RMP_PBAR_U2D` (grows up to down).

**`rmp_cnt_t Prog`**
Progress of the progress bar; must be between 0 and 100.

**`rmp_ptr_t Fore`**
Foreground colour of the progress bar, that is, the colour of the progress.

**`rmp_ptr_t Back`**
Background colour of the progress bar, that is, the colour of the empty region.

Returns

**`None`**
No return value.

### 4.5.14 Change the progress of a progress bar

#### RMP_Progbar_Set

Change progress of a progress bar

```
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)
```

This function changes the progress of an already drawn progress bar.

Parameters

**rmp_cnt_t Coord_X**
X coordinate of the progress bar's top-left corner.

**rmp_cnt_t Coord_Y**
Y coordinate of the progress bar's top-left corner.

**rmp_cnt_t Length**
Length of the progress bar.

**rmp_cnt_t Width**
Width of the progress bar.

**rmp_cnt_t Style**
Style of the progress bar; one of `RMP_PBAR_L2R`, `RMP_PBAR_D2U`, `RMP_PBAR_R2L`, or `RMP_PBAR_U2D`.

**rmp_cnt_t Old_Prog**
Old progress; must be between 0 and 100.

**rmp_cnt_t New_Prog**
New progress; must be between 0 and 100.

**rmp_ptr_t Fore**
Foreground colour of the progress bar.

**rmp_ptr_t Back**
Background colour of the progress bar.

#### Returns

**None**
No return value.

## 4.6 GUI design guidelines

Because this library is extremely limited compared with a full graphics library, some basic principles and design techniques must be followed when designing an interface. These techniques are described in the following sections.

### *4.6.1 Planar design*

Because this graphics library does not support a window manager, a planar design is preferred. Planar design is also the current design trend. For its concepts and details, refer to relevant websites and literature, such as Apple's UI design guidelines.

### *4.6.2 Use a simple background and monochrome bitmaps*

Because this GUI does not actually support layer stacking or semi-transparent blending, its anti-aliased bitmap function requires a fixed background colour for the calculation. If a non-monochrome background is used, the function cannot be used. In fact, most interfaces do not need a complex background image; a dark grey background is often sufficient.

At the same time, try to avoid graphics other than monochrome bitmaps. Greyscale and colour bitmaps consume a great deal of flash space. Monochrome bitmaps can save about 90% or more storage compared with grey or colour bitmaps.

### *4.6.3 Font-library design and implementation*

Two methods can be used to implement a font library with this GUI. The first is to make the characters used into monochrome bitmaps and store them in the microcontroller's ROM; the other is to use an

external font chip. For interfaces that require high visual quality but not a full character set, the first method is recommended; for those that require a full character set, the second method must be used.

### 4.6.4 Design and implementation of complex controls

Because this GUI provides only the most basic controls, and these basic controls often cannot meet the requirements of interfaces with higher visual standards, custom controls must be designed. To display a custom control, use the bitmap drawing functions provided by this GUI to draw it, and when a change in the control state is detected, update the control to a new bitmap.

## 4.7 Optimised implementation of low-level drawing

Unlike the frame buffers common on microprocessors, the entire memory of a microcontroller may not even be enough for one frame, so the accompanying screen often has its own display memory. Such a screen is also called an MCU screen. Every change of display content must transfer data to the screen through an interface, and these interfaces may even be low-performance serial interfaces such as SPI or IIC. In the worst case, these serial interfaces are even emulated with GPIO, making performance worse. Therefore, minimising the number of interface operations for the same drawing task is of the utmost importance. If the whole screen is updated every time, and the XY coordinates are set again for every point, drawing performance is greatly reduced.

The following methods can significantly improve the drawing speed of a microcontroller.

### 4.7.1 Region control

In common graphics frameworks, every redraw involves the whole screen or window, but the pixels that actually change are often only a very small part of the screen or window. It is therefore sufficient to update only the part that changed, without redrawing all elements every time. This method can be combined with the methods below to eliminate unnecessary screen updates as far as possible.

### 4.7.2 Macro inlining

`RMP_POINT` is called very frequently, so if it can be declared as a simple operation, it should not be declared as a function. This saves function call overhead and allows the compiler to perform deep optimisation. This method can be combined with the methods below to eliminate function call overhead as far as possible.

### 4.7.3 Collinear auto-increment

The RMP graphics interface uses only one low-level macro, `RMP_POINT`, but this does not mean that every drawing operation must be done point by point. Common drawing functions such as lines and rectangles tend to draw continuously in one direction, so the screen's drawing-coordinate auto-increment direction can be set before drawing. If the drawn points are all on the same row or column, and the coordinates passed to `RMP_POINT` are detected to be an extension of the previous coordinates in the auto-increment direction, there is no need to set the drawing coordinates again; only the drawing data needs to be transferred continuously.

### 4.7.4 Write combining

Some monochrome displays always write N consecutive pixels at a time and cannot write a single point without reading the screen. In this case, several `RMP_POINT` calls must be combined into one write: if consecutive `RMP_POINT` calls do not switch the drawing row, wait until all N pixels on the same line have

been written before writing to the screen. If the last transfer has fewer than N pixels, either (1) read back the colours of the remaining pixels from the screen to complete the group, or (2) fill the remaining pixels with the background colour when the background colour is known.

### 4.7.5 Block buffering

Although a microcontroller's memory cannot hold a complete frame buffer, it may be enough for part of the screen. A 320×240 16-bit true-colour screen needs 153 kB of memory for a complete buffer, but if only one quarter is buffered, 38.4 kB is enough. We can therefore draw the content for different blocks of the screen separately and upload each block to the screen at once. When drawing each block, the whole screen must be simulated, but only the part falling inside the block updates the block buffer. Because the microcontroller's computing speed is much higher than the I/O interface speed, simulating several full-screen draws may be worthwhile. For a screen of the same area, the more blocks there are, the less memory is used, but the more simulated full-screen redraws are needed; the block size must therefore be chosen as a compromise.

Some screens also support setting an update region, so when writing a complete block the coordinate setup can be omitted entirely, further improving screen write speed.

### 4.7.6 Palette method

For 16-bit or 32-bit true-colour screens, each pixel needs two or four bytes to express its colour. But the screen does not have to display every colour at all times; often the number of colours on one screen is small. In this case, the palette method can compress the buffer space. If only 16 colours are displayed on one screen, four bits per pixel are enough to express a colour code, where each colour code corresponds to a complete 16-bit or 32-bit colour. When writing data to the screen, look up the real colour data by colour code and then write that colour data to the screen. The lookup table can be changed at any time during program execution to use a different set of 16 colours.

This method can be combined with block buffering to reduce memory further. Suppose a TFT colour screen has a resolution of 320×240, one quarter is buffered at a time, and 16 colours are displayed; only 9.6 kB of memory is needed for efficient screen writing, and many 16-bit microcontrollers have memory of this order. Suppose an OLED mini colour screen has a resolution of 128×64, one quarter is buffered at a time, and 16 colours are displayed; only 1 kB of memory is needed for efficient screen writing, covering almost all 8-bit microcontrollers.

### 4.7.7 DMA transfer

When the interface supports DMA transfer, DMA can greatly reduce the CPU load. When DMA is available, it is recommended to always use DMA, especially for serial interfaces that generate frequent interrupts.

This method can be combined cleverly with block buffering; two block buffers are then needed. While the CPU performs a simulated full-screen drawing in one buffer, DMA can be started on the other buffer, so that DMA transfer and buffer updates proceed together. Because CPU drawing is generally faster than DMA transfer,[2] the buffer size can be chosen so that DMA transfer finishes slightly faster than the CPU's full-screen simulation. In this way, the microcontroller updates the screen at full speed as seen from the outside, and the CPU's full-screen drawing time is completely hidden, achieving the theoretical maximum performance.

However, this method cannot be combined with the palette method unless the DMA hardware can perform the palette lookup automatically.

[2] Serial interfaces often have low transfer rates.

## 4.8 Chapter bibliography

None.

# 5. Formal Verification

## 5.1 Introduction to formal verification

The essence of an embedded real-time operating system is complexity, diversity, and reliability. The hardware-independent part of RMP contains about 2000 lines of code, and its parts have complex interactions. An operating system must satisfy many different application requirements and guarantee functional correctness. At the same time, development efficiency, cost, and the development cycle must be controlled, and software delivery must be assured.

Traditional software design is based on natural-language thinking, design, and description; it is often one-sided and ambiguous, and easily misunderstood. It cannot be checked rigorously and can only be analysed by human reasoning. Semi-formal methods such as UML use relatively clear graphical descriptions, and some tools can generate code frameworks and check them automatically. Both methods test a system by designing a series of cases, at most with structured testing. Neither can guarantee that the system has no errors, and neither is suitable for the analysis and development of life-critical systems. A stricter development process is therefore needed. A fully formal method is based on precisely defined mathematical concepts and languages and can be checked and analysed by automated tools. It brings mathematical rigour to every stage of software development and uses rigorous mathematical proof to guarantee that the system has no defects.

The EAL standard for software security has seven levels,[1] described in Table 5-1.

Most operating systems, such as FreeRTOS, Windows, and Linux, are certified at EAL4. A few other systems are certified at EAL5, and INTEGRITY-178B is certified at EAL6+. Because RMP's structure is relatively simple, and its security targets include only functional correctness and not information security, it is easier for RMP to obtain a higher certification level.[2]

[1] EAL7+ is also included.
[2] Most CC/CAPP/LSPP design standards are not applicable.

Table 5-1. EAL levels and their descriptions.

| Level | Description |
|---|---|
| EAL1 | A functionally tested system. It provides a basic description of the security assurance targets, including the functional specification and a potential-error analysis. Every target is listed and evaluated by testing. Compared with an unevaluated system, EAL1 is only a small security improvement. |
| EAL2 | A system developed with basic software-engineering methodology and structured development. It provides a complete description of the security targets, including the basic functional specification and security architecture. Every target is listed and independently evaluated by testing; security testers with some attack capability must also attack the targets to confirm that no problem exists. EAL2 has stricter test specifications and system weakness analysis than EAL1. |
| EAL3 | A system systematically tested and checked with a strengthened software-engineering methodology. It provides a complete description of the security targets, including the basic functional specification and a detailed design of the security architecture. Every target is listed and independently evaluated by testing; testers with some attack capability must attack the detailed design of the security architecture. EAL3 has stricter system architecture specifications and corresponding tests than EAL2. |
| EAL4 | A system systematically designed, tested, and evaluated with a strengthened software-engineering methodology. It provides a complete description of the security targets, including the detailed functional specification and security architecture, and a division of some key modules with their implementation. Every target is listed and independently evaluated by testing; testers with above-average attack capability must attack the detailed design of the security architecture. EAL4 has stricter design descriptions, implementation review, and stronger tests than EAL3. |
| EAL5 | A system designed and tested with semi-formal methods. It provides a complete description of the security targets, including the detailed functional specification and security architecture, the division of all key modules described semi-formally, and the implementation of some modules. Every target is listed and independently evaluated by testing; testers with considerable attack capability must attack the detailed design of the security architecture. EAL5 has semi-formal design descriptions, a clearer architecture, and stronger tests than EAL4. |
| EAL6 | A system designed, tested, and verified with semi-formal methods. It provides a complete description of the security targets, including the detailed functional specification and security architecture, the division of all key modules, and the implementation of all modules. EAL6 requires a complete formal description of the security-target policy and a semi-formal description of its functions. The functional design must be modular, layered, and concise. Every target is listed and independently evaluated by testing; testers with strong attack capability must attack the detailed design of the security architecture. EAL6 has more comprehensive design analysis, weakness testing, and structured implementation descriptions than EAL5. |
| EAL7 | A system designed, tested, and verified with formal methods. It provides a complete description of the security targets, including the detailed functional specification and security architecture, the division of all key modules, and a structured implementation of all modules. EAL7 requires a complete formal description of the security-target policy and a semi-formal description of its functions. The functional design must be modular, layered, and concise. Every target is listed and subjected to complete independent evaluation testing against the implementation; testers with extremely strong attack capability must attack the detailed design of the security architecture. EAL7 has further formal design and more comprehensive tests than EAL6. |
| EAL7+ | A system designed, tested, and verified with strict formal methods. It provides a complete description of the security targets, including the detailed functional specification and security architecture, the division of all key modules, and a structured implementation of all modules. EAL7+ requires a complete formal description of the functions and policy of the security targets. The functional design must be modular, layered, and concise, and the implementation must be proved to conform completely to the formal description. Every target is listed and subjected to complete independent evaluation testing against the implementation; the test cases must be generated automatically by formal tools; testers with extremely strong attack capability must attack the detailed design of the security architecture. EAL7+ has further formal implementation and formal testing than EAL7. RMP is designed and tested according to the EAL7+ standard. |

## 5.2 Formal specification of the system

In RMP, the architecture-dependent part is very short[3] and easy to verify, so it is not formally verified. Another reason is that the architecture-dependent part changes with the chip, and RMP supports dozens of platforms; building a formal model for each one is not practical in terms of manpower. The main verification work therefore concentrates on the 1500 lines of code in `rmp_kernel.c`, about 1000 lines excluding comments. For the same multi-architecture reason, the correctness of this code is verified only at the level of C language semantics, not at the level of binary instructions. This means that the correctness of the final executable still depends on the correctness of the compiler. In practice, compilers usually have a high certification level, so trusting the compiler is reasonable. If the compiler is not to be trusted, the formally verified CompCert [2] compiler can be used.

Formal verification cannot completely eliminate erroneous operation of a software/hardware system, especially problems caused by hardware errors.[4] RMP's formal verification cannot prevent failures caused by hardware errors or by errors in the hardware abstraction layer (HAL).

## 5.3 Formal proof

This will be completed as future work.

## 5.4 Other documents

This will be completed as future work.

## 5.5 Chapter bibliography

[1] Common Criteria, "Common Criteria for information technology security evaluation, part 3: Security assurance components," Common Criteria, Technical Report, 2012.

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

[3] Usually within 50 lines of code.

[4] For example CPU or peripheral silicon errors; these are often described in the manufacturer's chip errata.

# 6. Porting RMP to a New Architecture

## 6.1 Porting overview

Porting an operating system means modifying it so that it can run on a new architecture and platform. Sometimes, making it possible to build the operating system with a new compiler is also called porting. Compared with the porting process of $\mu$C/OS and RT-Thread, porting RMP is very simple. All RMP code is written in ANSI/ISO C89, relatively close to MISRA C, and contains a minimal amount of assembly. The porting work therefore consists of only a few steps.

Before porting, some preparation is needed to confirm that porting is possible. Then the corresponding code is written for each part. Finally, test cases can be used to check that the port is successful. Because the architecture-independent part of RMP has been formally verified, no modification to that part is permitted; otherwise the system's certification is destroyed.

The low-level interface of the graphics library is described in Chapter 4 and is not repeated here.

## 6.2 Checks before porting

### 6.2.1 Processor

RMP requires the selected processor to have at least two interrupt vectors and one timer. Apart from this, RMP imposes no other requirements on the processor. RMP cannot run on a platform with less than 8 KiB of flash memory, and it cannot run on a processor narrower than 16 bits. To run an operating system on such platforms, the state-machine-based RMS may be a better choice. If the selected processor is multi-core or has an MMU, the microkernel-based RME may be a better choice. RMP does not support a hardware stack mechanism; the stack must be implemented in software and cannot be implemented in hardware inside the processor.[1][2]

### 6.2.2 Compiler

RMP requires the compiler to be C89 compliant and to generate code according to a function calling convention. Because RMP code is very standard and does not use any C runtime library functions, any compiler that conforms to ANSI C89 is sufficient. Common compilers such as GCC, CLANG/LLVM, MSVC, ARMCC, ICC, IAR, and TASKING all satisfy this requirement. RMP does not use compiler extensions that vary between implementations, such as bitfields, `enum`, and structure packing, and it avoids undefined behaviour in C as far as possible, so maximum compatibility is guaranteed.

When using the compiler, care should be taken to disable dead-code elimination and link-time optimisation on low-quality compilers; it is also preferable to disable loop-invariant hoisting on such compilers. Do not use aggressive optimisation options. On ordinary compilers, the recommended optimisation level is GCC `-O2` or an equivalent level.

[1] That is, the stack pointer and stack contents can be manipulated by software.
[2] This is typical of a few architectures such as PIC microcontrollers.

### 6.2.3 Assembler

RMP requires the assembler to import symbols from C and call them according to the calling convention. It also requires the code produced by the assembler to be exported and callable from C according to the calling convention. These requirements are usually easy to satisfy. If the compiler supports inline assembly, a separate assembler is not needed.

### 6.2.4 Debugger

RMP has no special requirements for the debugger. A debugger is best if available, but porting is possible without one. With a debugger, kernel variables can be inspected directly. Without a debugger, the lowest-level `RMP_Putchar` function should be implemented first to print individual characters; the log can then be printed with this facility. The implementation of this function is described in the next section.

## 6.3 Architecture-dependent parts of RMP

All architecture-dependent source files are placed under the corresponding architecture name in the `Platform` folder. For example, the ARMv7-M architecture is under `Platform/A7M`, and its headers are under `Include/Platform/A7M`; other architectures follow the same rule.

Each architecture contains one or more source files and one or more header files. When the kernel includes an architecture-dependent header, it always includes `Include/rmp_platform.h`, which in turn includes the top-level header of the corresponding architecture. When changing RMP's build target, this file is modified to include the headers of the new target. For example, to compile for ARMv7-M, this header should include all the low-level headers corresponding to ARMv7-M.

When porting to another architecture, the ARMv7-M low-level implementation can be used as a template and the new port can be developed from it.

### 6.3.1 Type definitions

The first part to port for each architecture and compiler is the type definitions. For these, it is only necessary to determine how the processor's word length is expressed by the compiler and to define it with `typedef`. Note that for some architectures and compilers, `long` corresponds to two machine words rather than one; in that case `int` should be used to represent one machine word. For other architectures and compilers, `int` is half a machine word and `long` is one machine word; in that case `long` should be used for one machine word.

When necessary, small programs using `sizeof()` can be written to determine the compiler's word model. To make the low-level code easier to write, the fixed-width integer types in Table 6-1 are recommended. When defining them, the lengths of the compiler's `char`, `short`, `int`, and `long` types must also be determined.

RMP requires the three types in Table 6-2.

### 6.3.2 Macro definitions

The second part is the macro definitions. The macros are listed in Table 6-3.

Table 6-1. Overview of typedefs.

| Type | Typical definition | Meaning |
|---|---|---|
| `rmp_s8_t` | `typedef signed char` | Signed 8-bit integer. |
| `rmp_s16_t` | `typedef signed short` | Signed 16-bit integer. |
| `rmp_s32_t` | `typedef signed int` | Signed 32-bit integer. |
| `rmp_u8_t` | `typedef unsigned char` | Unsigned 8-bit integer. |
| `rmp_u16_t` | `typedef unsigned short` | Unsigned 16-bit integer. |
| `rmp_u32_t` | `typedef unsigned int` | Unsigned 32-bit integer. |

Table 6-2. Overview of necessary typedefs.

| Type | Typical definition | Meaning |
|---|---|---|
| `rmp_ptr_t` | `typedef unsigned long` | Pointer integer type; an unsigned integer of the same length as one machine word. |
| `rmp_cnt_t` | `typedef long` | Counting-variable type; a signed integer no longer than one machine word but at least 32 bits. |
| `rmp_ret_t` | `typedef long` | Return-value type; a signed integer of the same length as one machine word. |

Table 6-3. Overview of macro definitions.

| Macro | Typical | Meaning |
|---|---|---|
| `RMP_EXTERN` | `extern` | The compiler's `extern` keyword. When using a C++ compiler, it can be defined as `extern "C"` to prevent name mangling. |
| `RMP_WORD_ORDER` | 5 | The power of two corresponding to the processor word length.[3] For example, 8-bit[4] and 16-bit processors use 4, 32-bit processors use 5, and 64-bit processors use 6. |
| `RMP_STACK_TYPE` | FULL_DESCEND | Stack type. Use `RMP_STACK_FULL_ASCEND` for a full ascending stack, `RMP_STACK_FULL_DESCEND` for a full descending stack, `RMP_STACK_EMPTY_ASCEND` for an empty ascending stack, and `RMP_STACK_EMPTY_DESCEND` for an empty descending stack. |

[3] Measured in bits.
[4] An 8-bit processor has at least 16-bit pointers, so RMP regards it as a 16-bit processor.

| Macro | Typical | Meaning |
|---|---|---|
| `RMP_STACK_ALIGN` | 3 | Stack alignment requirement. If the value is $X$, the stack is aligned to $2^X$ bytes when a thread is entered. |
| `RMP_STACK_ELEM` | `rmp_ptr_t` | Element type inside the stack; the type pushed on each normal data push. |
| `RMP_STACK_STRUCT` | `struct RMP_XXX_Stack` | Name of the structure used to initialise a thread stack. |
| `RMP_INIT_STACK_SIZE` | 1024 | Initial thread stack size in bytes. The initial stack is an array declared inside `rmp_kernel.c`; if an architecture has special address requirements for the stack,[5] the kernel data section must be linked at a suitable address. |
| `RMP_PREEMPT_PRIO_NUM` | 32 | Maximum number of preemptive priorities; must be at least 3. In practice, if memory is not extremely tight, a multiple of the processor word length[6] is recommended, but this is not required. Usually it is sufficient to set this to the word length. |
| `RMP_SLICE_MAX` | 100000 | Maximum thread time slice or delay slice; must not exceed $2^{N-1} - 1$ (that is, `RMP_MASK_INTMAX`), where $N$ is `RMP_WORD_ORDER`. |
| `RMP_SEM_CNT_MAX` | 1000 | Maximum semaphore count; must not exceed $2^{N-1} - 1$ (`RMP_MASK_INTMAX`). |

[5] For example, it must reside in the low 64 KiB.
[6] Measured in bits.

| Macro | Typical | Meaning |
|---|---|---|
| `RMP_INT_MASK()` | `MASK(SYSPRIO)` | Whether to mask interrupts that can call the interrupt send functions when the scheduler is locked. If this feature is not used, the macros may be empty; interrupts that call the send functions can then occur while the scheduler is locked, but their send functions will fail. If the feature is used, `RMP_INT_MASK()` can mask all system interrupts and `RMP_INT_UNMASK()` can unmask them. A global interrupt disable is also possible but affects real-time performance. `MASK(SYSPRIO)` masks all interrupts at or below the `SYSPRIO` priority, and `MASK(0x00)` removes the mask; the `SYSPRIO` value should be the highest priority of an interrupt that can call the send functions. |
| `RMP_INT_UNMASK()` | `MASK(0x00)` | See `RMP_INT_MASK()`. |
| `RMP_MSB_GET(VAL)` | `32-__builtin_clz(VAL)` | Index of the highest binary bit, counting the lowest bit as 0. Many processors provide a count-leading-zeros instruction or builtin; otherwise use `RMP_MSB_Generic`. |
| `RMP_LSB_GET(VAL)` | `__builtin_ctz(VAL)` | Index of the lowest binary bit, counting from 0. Many processors provide a count-trailing-zeros instruction or builtin; otherwise use `RMP_LSB_Generic`. |
| `RMP_STKSEG_ENABLE` | 0 | Whether to save and restore a stack segment register in addition. If enabled, RMP defines `RMP_SS_Cur` for the low-level assembly. Most architectures do not need this. |
| `RMP_ASSERT_ENABLE` | 1 | Whether to enable the assertion macro. Define as 1 to detect errors with `RMP_ASSERT`; define as 0 to skip assertions. |

| Macro | Typical | Meaning |
|---|---|---|
| `RMP_CHECK_ENABLE` | 1 | Whether to enable parameter checking. If 0, system calls do not check argument validity and, except where specially noted, do not return error values. Keep it enabled unless performance or code size is extremely constrained. |
| `RMP_DBGLOG_ENABLE` | 1 | Whether to enable kernel print macros. If 0, the macros are disabled. |

### 6.3.3 Assembly functions and macros

RMP requires only five short low-level assembly functions or macros, implemented in assembly or inline assembly. They are listed in Table 6-4.

**Table 6-4.** Overview of low-level assembly functions and macros.

| Function | Meaning |
|---|---|
| `RMP_Int_Disable` | Disable processor interrupts. |
| `RMP_Int_Enable` | Enable processor interrupts. |
| `_RMP_Start` | Start the initial thread. |
| `RMP_YIELD` | Trigger a thread switch. |
| `RMP_YIELD_ISR` | Trigger a thread switch from an interrupt. |

The implementation methods and requirements are described later.

### 6.3.4 System interrupt vectors

RMP requires at least two interrupt vectors, implemented in assembly or inline assembly, listed in Table 6-5.

### 6.3.5 Other low-level functions

These functions concern startup, debugging, and other low-level aspects. They may be written in assembly, in C, or partly in each. They are listed in Table 6-6.

Table 6-5. Overview of system interrupt vectors.

| Vector | Meaning |
|---|---|
| `System timer vector` | Handle the system timer interrupt and manage time-slice usage. |
| `Thread context-switch vector` | Perform thread context switches. |

Table 6-6. Overview of other low-level functions.

| Function | Meaning |
|---|---|
| `RMP_Putchar` | Print one character to the kernel debugging console. |
| `_RMP_Lowlvl_Init` | Initialise the low-level hardware. |
| `_RMP_Stack_Init` | Initialise a thread's stack. |

## 6.4 Porting the assembly functions and macros

The prototypes and porting requirements of the assembly functions and macros are described in detail below.

### 6.4.1 Implementation of RMP_Int_Disable

#### RMP_Int_Disable

Disable interrupts

```
void RMP_Int_Disable(void)
```

This function disables the processor's interrupts and returns.

**Parameters**

**None**
No parameters.

**Returns**

**None**
No return value.

This function only needs to disable processor interrupts and return. There are no special precautions; in most cases it is a single CPU register or peripheral write followed by a return.

### 6.4.2 Implementation of RMP_Int_Enable

#### RMP_Int_Enable

Enable interrupts

```
void RMP_Int_Enable(void)
```

This function enables the processor's interrupts and returns.

**Parameters**

**None**
No parameters.

**Returns**

**None**
No return value.

This function only needs to enable processor interrupts and return. There are no special precautions; in most cases it is a single CPU register or peripheral write followed by a return.

### 6.4.3 Implementation of _RMP_Start

#### _RMP_Start — Start the initial thread

```
void _RMP_Start(rmp_ptr_t Entry, rmp_ptr_t Stack)
```

This function switches from the kernel stack to the thread stack and is called only at the end of system startup; after this, the system enters normal operation. It only needs to assign `Stack` to the stack pointer and jump to `Entry`; it never returns.

**Parameters**

**`rmp_ptr_t Entry`**
Entry address of the initial thread, actually `RMP_Init`.

**`rmp_ptr_t Stack`**
Stack address of the initial thread.

**Returns**

**None**
No return value.

This function switches from the kernel stack to the thread stack and is called only at the end of the system boot process. After it, the system enters normal operation. It only needs to assign `Stack` to the stack pointer and jump directly to `Entry`; it never returns.

### 6.4.4 Implementation of RMP_YIELD

#### RMP_YIELD — Request a context switch

```
void RMP_YIELD(void)
```

This macro notifies the hardware abstraction layer to perform a context switch. It must guarantee that a context switch is arranged after it is called and before thread code continues. Normally it software-triggers a pendable thread context-switch vector by writing a memory address or executing a special instruction. If interrupts are not masked, the software interrupt fires before the macro returns; if interrupts are disabled, the software interrupt must remain pending and fire before the interrupt-enable function returns. The macro must contain a compiler barrier so that the compiler does not optimise the following code before it.[a]

**Parameters**

**None**
No parameters.

**Returns**

**None**
No return value.

[a] Link-time optimisation builds need particular care here.

### 6.4.5 Implementation of RMP_YIELD_ISR

#### RMP_YIELD_ISR

Request a context switch from an ISR

```
void RMP_YIELD_ISR(void)
```

This macro is similar to `RMP_YIELD`, but it is called only by the interrupt send functions whose names end in `_ISR`. It also software-triggers the pendable context-switch vector, and its implementation is identical to `RMP_YIELD`. The two macros are separate so that a port can adopt another method; see Section 7.1.2.

**Parameters**

**`None`**
No parameters.

**Returns**

**`None`**
No return value.

## 6.5 Porting the system interrupt vectors

RMP requires two interrupt vectors to be ported: the system timer interrupt vector and the thread context-switch vector. The timer vector need not be written in assembly; the context-switch vector must be.

### 6.5.1 System tick vector

The timer interrupt vector only needs to call the timer handler `_RMP_Tim_Handler(Slice)`. In a ticked kernel, one time slice always elapses on each timer interrupt, so `Slice` is 1. In a tickless kernel, it is the slice value last programmed into the timer. The value is not limited by `RMP_SLICE_MAX`, but it must not exceed $2^{N-1}-1$ (`RMP_MASK_INTMAX`). The handler is implemented by the system and need not be written by the user; see Section 7.1.1 for tickless operation and `_RMP_Tim_Elapse`.

#### _RMP_Tim_Handler

Timer interrupt handler

```
void _RMP_Tim_Handler(rmp_ptr_t Slice)
```

Executes timer interrupt handling.

**Parameters**

**`rmp_ptr_t Slice`**
In a ticked kernel, one time slice always elapses per clock interrupt, so pass 1; in a tickless kernel, pass the slice value last programmed into the timer. The number is not limited by `RMP_SLICE_MAX`, but it must not exceed $2^{N-1}-1$ (`RMP_MASK_INTMAX`), where $N$ is `RMP_WORD_ORDER`.

**Returns**

**`None`**
No return value.

### 6.5.2 Context-switch vector

The context-switch interrupt vector must be written in assembly and must perform the following steps in order:

1. push all the CPU's basic registers onto the thread stack;
2. store the current thread stack pointer in `RMP_SP_Cur`;
3. switch to the kernel stack;[7]
4. call `_RMP_Run_High` to switch threads;
5. assign `RMP_SP_Cur` to the current thread stack pointer;
6. restore all the CPU's basic registers from the thread stack; and
7. switch to the thread stack and exit the interrupt.

`_RMP_Run_High` is implemented by the system and need not be written by the user. A thread context switch initiated from a user thread[8] may also use inline assembly, in which case interrupts must be disabled during the switch; see Section 7.1.2.

#### _RMP_Run_High — Thread context-switch interrupt handler

```
void _RMP_Run_High(void)
```

Executes thread context-switch handling. The switch updates the variables used by the context-switch assembly section and by other kernel functions.

**Parameters**

**None**
No parameters.

**Returns**

**None**
No return value.

## 6.6 Porting the other low-level functions

Of the remaining low-level functions, `RMP_Putchar` and `RMP_MSB_Get` were discussed in Chapter 2. Only `_RMP_Lowlvl_Init` and `_RMP_Stack_Init` are discussed here.

### 6.6.1 Low-level hardware initialisation

This function initialises all low-level hardware. When initialising the interrupt system, the system tick timer should be given the lowest priority and the thread context-switch interrupt the second-lowest priority; neither may nest with any other interrupt. Interrupt vectors that call `RMP_Thd_Snd_ISR` or `RMP_Sem_Post_ISR` must also not nest with other interrupt vectors. All other interrupt vectors may nest freely.

#### _RMP_Lowlvl_Init — Initialise low-level hardware

[7] This step is optional, but a separate kernel stack and thread stack are recommended.
[8] Rather than from an interrupt vector.

```
void _RMP_Lowlvl_Init(void)
```

Initialises all basic system hardware, including the PLL, CPU, interrupt system, and system tick timer.

#### Parameters

**None**
No parameters.

#### Returns

**None**
No return value.

### 6.6.2 *Thread stack initialisation*

This function is called by the thread creation function. Because the second half of the context-switch interrupt pops registers from this stack, the registers must be placed in the same order, especially the thread entry and its parameter. The exact sequence depends on the processor and on the push/pop order of the context-switch interrupt.

In general, the function selects an address at the head[9] or tail[10] of the stack according to the processor's stack-growth direction and initialises it there. If the processor has a stack alignment requirement,[11] the function must also ensure that the stack pointer satisfies that alignment after the pops. The initial values of sensitive registers[12] must conform to the architecture's binary interface standard to prevent errors when the thread starts.

To make the function easier to write, RMP provides the macro `RMP_STACK_CALC(PTR, CTX, STK, SZ)`. It takes the stack start `STK` and size `SZ` and returns the real initial stack pointer `PTR` and a pointer `CTX` to the context structure used for initialisation. `PTR` can be used as the function's return value, and filling the structure pointed to by `CTX` completes the stack initialisation.

#### _RMP_Stack_Init — Initialise a thread stack

```
rmp_ptr_t _RMP_Stack_Init(rmp_ptr_t Stack, rmp_ptr_t Size, rmp_ptr_t Entry, rmp_ptr_t
Param)
```

Fills the thread stack so that the new thread, on its first run, behaves as if it had returned to this point.

#### Parameters

**`rmp_ptr_t Stack`**
Start address of the thread's stack area.

**`rmp_ptr_t Size`**
Size of the thread's stack area.

**`rmp_ptr_t Entry`**
Thread entry point.

[9] Ascending stack.
[10] Descending stack.
[11] For example ARMv6-M and ARMv7-M.
[12] Such as segment registers and condition-code registers. The toolchain's initial values can be obtained by entering debug mode and stopping at the entry to `main`, or captured in `_RMP_Start`. For a stack segment register, see Section 7.5.3.2.

**`rmp_ptr_t Param`**
Parameter passed to the thread.

### Returns

**`rmp_ptr_t`**
The real start address of the thread stack. The thread will pop from here on its first run.

## 6.7 Chapter bibliography

None.

# 7. Appendix

## 7.1 Implementation of special kernel features

Some features of RMP are not implemented by the kernel by default. It is possible to implement them in RMP; the principles are described below. If the user needs these features, they must be implemented by the user.

### 7.1.1 Implementation of a tickless kernel

A tickless kernel requires the system to have a timer with a long duration and high precision, and to generate variable scheduler time interrupts from that timer. The ratio between the time slice and the timer reload value can be freely chosen by the implementer. To make a tickless kernel easier to implement, the system provides two functions, `_RMP_Tim_Future` and `_RMP_Tim_Elapse`; if a tickless kernel is not implemented, they can be ignored.

To make the timer generate an interrupt at the nearest potential timeout, after every timer interrupt,[1] context switch,[2] or new thread entering the delay queue,[3] the current thread's remaining time must be checked, then the remaining time of the earliest delayed thread, and the smaller of the two must be chosen as the next hardware timer timeout.

The kernel provides `_RMP_Tim_Future` to obtain this timeout. Two special cases must be handled: (1) if the timeout is 0, a thread delay has expired and a timer interrupt must be triggered immediately; (2) if the interval is longer than the timer can support, simply set the timer timeout to the maximum allowed value. RMP's design guarantees that only the time that actually elapsed is deducted from a thread's remaining time, so there is no need to worry about losing delay time.

#### _RMP_Tim_Future

Get the nearest timeout in slices

```
rmp_ptr_t _RMP_Tim_Future(void)
```

Obtains the number of time slices until the nearest future timeout, which can be used to set the next timeout of the system timer.

**Parameters**

**None**

No parameters.

**Returns**

[1] For example the timer interrupt hook `RMP_TIM_HOOK`. A timer interrupt can mean (1) a thread has exhausted its time slice, (2) a thread delay has expired, or (3) the hardware timer's maximum configurable timeout is not long enough and the delay must be completed in several parts. In case (1) a context switch always occurs; in case (2) a context switch may occur; in case (3) no context switch can occur. When a context switch is detected as pending, this timer programming is not needed, because the following context switch will do it.

[2] For example the context-switch hook `RMP_SCHED_HOOK`.

[3] For example the delay-insertion hook `RMP_DLY_HOOK`.

**rmp_ptr_t**
Number of time slices until the nearest future timeout.

Before `_RMP_Tim_Future` calculates the remaining time of the nearest delayed thread, it is necessary to ensure that the timestamp `RMP_Timestamp` is up to date. Otherwise the calculation will be wrong. The kernel therefore provides `_RMP_Tim_Elapse` to update the timestamp. Outside a timer interrupt,[4] it must be called before `_RMP_Tim_Future`; inside a timer interrupt it need not be called, because the timer interrupt itself updates the timestamp.

Note: observing where `_RMP_Tim_Elapse` is called shows that it appears either in the context-switch vector or in `_RMP_Dly_Ins`, which is necessarily followed by a context switch; RMP always refills a thread's time slice after a context switch.[5] Therefore `_RMP_Tim_Elapse` does not need to update the current thread's remaining slice count. This further means that in a context switch, `_RMP_Tim_Elapse` only needs to be called before `_RMP_Tim_Future`, and does not have to be fixed before the switch.[6]

### _RMP_Tim_Elapse

Record elapsed time

```
void _RMP_Tim_Elapse(rmp_ptr_t Slice)
```

Records the number of time slices the current thread has used before calculating the timer timeout.

#### Parameters

**rmp_ptr_t Slice**
Number of slices elapsed since the timer was last reloaded. This can be obtained by reading the timer's counter register and computing its difference from the reload value. The number is not limited by `RMP_SLICE_MAX`, but must not exceed $2^{N-1}-1$ (`RMP_MASK_INTMAX`). The read may stop the timer first, because the next `RMP_SCHED_HOOK` necessarily reprograms it and it does not matter whether the timer keeps running after the read. After calling this function, `RMP_DLY_HOOK` must also clear the timer reload and count values, to avoid a duplicate time deduction by a later `RMP_SCHED_HOOK`.

#### Returns

**None**
No return value.

To save more energy, when only the initial idle thread is running and no thread is delayed, even the clock source of the system timer can be stopped. The system provides `_RMP_Tim_Idle` to detect this condition. To use this feature, interrupts must first be disabled, then the function is used to detect whether the timer clock source can be stopped, and finally interrupts are enabled atomically while entering an appropriate low-power state.[7] A concrete example is the CCS-MSP430G2553 project in the MSP430 port.

### _RMP_Tim_Idle

Query the system timer idle state

```
rmp_ret_t _RMP_Tim_Idle(void)
```

Determines whether the system timer and its clock source can be completely stopped to save energy.

[4] That is, in `RMP_SCHED_HOOK` and `RMP_DLY_HOOK`, but not in `RMP_TIM_HOOK`.
[5] In RME the slice is not refilled automatically; the user-mode scheduler must pass it manually.
[6] In RME the same-named function must be called at the beginning of the vector; otherwise, once the switch is complete, the consumed slice is charged to another thread.
[7] If the CPU does not provide an operation that atomically enables interrupts and stops the timer clock source, or if the clock source cannot start automatically when an interrupt arrives, this feature cannot be used.

#### Parameters

**`None`**
No parameters.

#### Returns

**`0`**
The timer and its clock source may be stopped.

**`-1`**
They may not be stopped.

It is worth noting that although a tickless kernel has lower CPU power consumption, its real-time behaviour is not necessarily better, because it necessarily includes time-elapse accounting before a context switch and timer reprogramming after it. Both can increase the length of the context-switch critical section and negatively affect the worst-case execution time. In a virtualised environment these disadvantages are multiplied. Therefore, a ticked kernel is recommended in non-low-power environments.

### *7.1.2 Saving and restoring thread context*

Normally, the system uses a dedicated thread context-switch interrupt vector, and performs thread context switching only in that vector. This makes the context-switch vector the only interrupt vector in the system that must be written in assembly, greatly simplifying porting. It also reduces the interrupt latency of other vectors, because entering and leaving other vectors does not require pushing and popping the full context. In addition, when several interrupts execute back-to-back, context saving, restoring, and switching are performed only after the last interrupt exits, saving a great deal of overhead. For these reasons, this method is recommended.

However, this method is not without disadvantages. It (1) requires an extra software-triggerable, pendable interrupt vector,[8] (2) requires that vector to be entered almost immediately after the software trigger,[9] (3) requires interrupts to be able to tail-chain directly, that is, the next interrupt must be able to follow the exit of the previous one without any main-program instruction in between,[10] (4) increases the latency of activating a thread from an interrupt, and (5) increases the latency of initiating a context switch from a thread. When conditions (1)–(3) are not met, or when (4) and (5) are not desired, the implementation ideas below can be used.[11]

First, write `RMP_YIELD` in assembly and perform the following steps:

1. disable interrupts;
2. simulate the processor's interrupt entry, pushing the registers that an interrupt would push, and ensure that interrupts are enabled in the pushed frame;
3. push all the CPU's basic registers onto the thread stack;[12]
4. store the current thread stack pointer in `RMP_SP_Cur`;
5. switch to the kernel stack;[13]
6. call `_RMP_Run_High` to switch threads;

[8] For example ARMv7-M PendSV.
[9] The vector must be entered immediately after the trigger, or the number of instructions between the trigger and entry must be controllable.
[10] Some CPUs must execute one main-program instruction after an interrupt return before accepting the next interrupt.
[11] For example when threads are used as cooperative fibers or coroutines, or when the software interrupt latency is high.
[12] The instruction pointer PC (or IP) needs special treatment: its original value must not be pushed; the return point of step 10 must be pushed instead.
[13] This step is optional, but a separate kernel stack and thread stack are recommended.

7. assign `RMP_SP_Cur` to the current thread stack pointer and switch to the thread stack;
8. pop all the CPU's basic registers from the thread stack;
9. call the interrupt-return instruction to enable interrupts and return; and
10. call the normal return instruction to return to the original program flow.

Then write the interrupt vector entry and exit in assembly. Add register saving (steps 3, 4, and 5 above) at every entry and register restoring (steps 7, 8, and 9 above) at every exit, and before the exit check whether `RMP_Sched_Pend` is set. If it is set, perform the thread switch (step 6). In this case the `RMP_YIELD_ISR` macro need not be defined.

When using this method, the context produced by `RMP_YIELD` must be restorable by the interrupt vector's exit, and the context produced by the vector's entry must be restorable by `RMP_YIELD`, especially on architectures that perform hardware pushes on interrupt entry. In addition, some architectures have long-running instructions[14] that the processor designer allows interrupts to break, recording their execution state so that they can be restored correctly on interrupt exit. For such architectures, `RMP_YIELD` must be able to resume such an instruction from the point where it was interrupted, not restart the whole instruction; otherwise the program logic will fail.[15] If resumption from the middle cannot be guaranteed but this method must be used, the port can lock the scheduler and use `RMP_Thd_Peek_ISR` to examine the next thread's context in advance; if that context is in the middle of an interrupted instruction and cannot be restored by `RMP_YIELD`, the slower interrupt-context-switch path is taken.[16]

For architectures without a dedicated context-switch interrupt, all official RMP ports use this method and package the assembly code into a macro. All system interrupts[17] must use this macro to save and restore context, while transparent interrupts[18] have no such requirement. See the MSP430 port for a concrete example.

This method can be further optimised by the measures described in the following subsections. None of the official RMP ports implement these optimisations, because after comprehensive consideration they are not worthwhile.

#### 7.1.2.1 Tolerating a software interrupt that does not enter immediately

A loop waiting for `RMP_Sched_Pend` to clear can be added after every call to `RMP_YIELD`, and code can continue only after it clears, to avoid the requirement in (2) that the vector enter almost immediately. The other requirements still cannot be easily avoided. Therefore, the official RMP ports for architectures that do not satisfy (1)–(3) do not use this method.

#### 7.1.2.2 Deferring interrupt disabling to improve real-time behaviour

Interrupt disabling can be deferred until after the registers are pushed, reducing the total disabled time. But if an interrupt arrives immediately after the push[19] caused by `RMP_Thd_Yield`'s `RMP_YIELD`, the thread context is saved twice. Similarly, enabling interrupts can be moved before the pops, but this is even more dangerous: an interrupt that breaks the `RMP_YIELD` pop sequence pushes another frame, and that frame may still be popped by `RMP_YIELD` in the future; however, `RMP_YIELD` may be interrupted again before it finishes popping, causing theoretically unbounded stack use.[20] To avoid these problems, official RMP ports always disable interrupts before pushing registers and do not enable them until the pops are complete.

[14] For example the ARMv7-M LDM, STM, and IT blocks, and the hardware loop instructions of some DSP processors.
[15] For example double pushes or incorrect condition evaluation.
[16] For example the ARMv7-M RVM port.
[17] System interrupts are those that call the `_ISR` series of functions or access an operating-system critical section.
[18] Transparent interrupts are those that do not call the `_ISR` series of functions and are unrelated to the operating system.
[19] Calls to `RMP_YIELD` caused by `RMP_Sched_Unlock` do not have this problem, because the scheduler is already locked and interrupts are disabled.
[20] In practice this is almost impossible, so it can be ignored in non-serious contexts.

#### 7.1.2.3 Using the calling convention to optimise interrupt latency

An interrupt vector does not necessarily request a context switch on every trigger, so aggressively saving and restoring all registers at vector entry and exit is unnecessary. Two optimisations are possible: (1) at entry save only the caller-saved registers required by the calling convention, execute the vector immediately, and at the end check `RMP_Sched_Pend` to decide whether a context switch is needed; if so, push the remaining registers and switch, otherwise pop the caller-saved registers and exit the interrupt; (2) further, at the end of the vector, determine whether other interrupts are pending after the exit; if so, defer the above decision until the last vector exits, and do not switch on this occasion regardless of `RMP_Sched_Pend`.[21] However, these two optimisations (1) must adapt to potentially different calling conventions of different compilers, (2) increase the complexity of the hardware abstraction layer, and (3) improve only the average efficiency while worsening the worst-case execution time; therefore they are not implemented in official RMP ports.

#### 7.1.2.4 Using the calling convention to optimise yield efficiency

Because `RMP_YIELD` is a macro that is really an assembly function, the compiler necessarily follows the calling convention before calling it, saving the caller-saved registers in advance and restoring them automatically after the function returns. In fact, `RMP_YIELD` only needs to save and restore the callee-saved registers; this can improve the efficiency of voluntary processor yielding between threads, a design that is quite common in monolithic kernels. However, the context produced by `RMP_YIELD` is then no longer complete, and the interrupt vector's context saving and restoring must be made compatible with it. Whatever compatibility method is used, interrupt latency will undoubtedly worsen, so this optimisation is not implemented in official RMP ports.

Note 1: A clever compatibility method is to let the interrupt vector use the thread stack and modify the interrupt-return path used for a context switch so that it (1) lets the processor acknowledge the interrupt early[22] but keeps interrupts disabled, (2) calls a variant of `RMP_YIELD` with a final state of interrupts disabled[23] to produce an incomplete context that can be restored by normal `RMP_YIELD`, and finally (3) pops the stack and exits the interrupt. Combined with the optimisation of Section 7.1.2.3,[24] the interrupt entry and exit then only push and pop caller-saved registers, and all callee-saved register saving and restoring is handled by `RMP_YIELD`, bypassing the compatibility problem. The cost is that (1) every thread stack must reserve space for the interrupt vector, increasing memory consumption, and (2) when switching back, an incomplete stack must be popped before the interrupt stack, adding several jumps and worsening the worst-case execution time.

Note 2: In the compatibility method above, some implementations go further to minimise interrupt-disabled time. They leave interrupts enabled after (1) the processor acknowledges the interrupt, so that accumulated pending interrupts can be serviced before `RMP_YIELD` is called, and they call the original `RMP_YIELD` directly at (2), so that interrupts can be serviced before the interrupt-exit pops. This does reduce interrupt-disabled time, but the thread may be interrupted again and have its registers pushed before `RMP_YIELD` has a chance to run or just after it finishes, causing the theoretically unbounded stack use described in Section 7.1.2.2. If the optimisation of Section 7.1.2.2 is also applied, stack use becomes even worse.

#### 7.1.2.5 Reusing code between yield and preemption

In official RMP ports, `RMP_YIELD` and the interrupt vector entry and exit always use two independent implementations, however similar they are, and no code is reused. The reasons are: (1) not all assemblers

[21] To achieve an effect similar to ARMv7-M PendSV.

[22] On processors such as ARMv7-M that record interrupt state and automatically send an end-of-interrupt (EOI) signal, this is required.

[23] It should not normally be called from an interrupt vector, and it always enables interrupts at the end.

[24] Of course it is also possible not to combine them, at the cost of pushing and popping the full stack, which unnecessarily worsens performance.

allow including assembly header files; where allowed, an `.inc` file is provided for the user to include in the interrupt vector assembly file and in the architecture's low-level assembly file, allowing reuse; where not allowed, a `.cpy` file is provided for the user to paste manually into the vector assembly file, and reuse is impossible; (2) the two implementations differ greatly on some architectures, so reuse is impossible; for example, some architectures perform special stack-alignment operations in hardware on interrupt entry and exit, which `RMP_YIELD` must simulate in another way.[25] Overall, rather than using a shared implementation for only some architectures and toolchains, it is better to use two independent implementations for all architectures; this keeps the low-level mental model uniform.

### 7.1.3 Saving and restoring coprocessor context

A coprocessor is a CPU extension module such as a floating-point unit or a vector processor; it also has its own context. If several threads share the same coprocessor concurrently, its context must be saved and restored together with the thread context during a context switch. RMP therefore provides two hooks, `RMP_CTX_SAVE` and `RMP_CTX_LOAD`. In the save hook, first detect whether the current thread used the coprocessor, and save its context if it did; in the load hook, also detect whether the coprocessor was used, and restore its context if it was. If it is impossible to detect whether the coprocessor was used, its context must always be saved and restored.

Because these two hooks are called after entering the kernel's C environment, on ports with a separate kernel stack they can use `RMP_SP_Cur` to save the context on the thread stack; all official RMP ports use a separate kernel stack. On ports that share the kernel stack with the thread, the coprocessor context must be saved in an additional structure; otherwise it may overwrite the kernel's return address and cause errors.

For most architectures, the official RMP ports already save and restore the coprocessor context on the stack in assembly. For these architectures, the floating-point register context can be saved and restored without the hooks. Different floating-point processor configurations may require different save/restore routines to be referenced in the assembly file.

Note 1: In official RMP ports, all context including the general registers is placed on the thread stack. This is because (1) almost all processors push at least the PC and FLAGS on interrupt entry, and some also force-push some general registers; putting all context into a structure would require copying these registers into the structure, which is bad for performance, and (2) if context that is not automatically pushed is saved separately into a structure, the PC and FLAGS in the thread stack and the structure would both need to be initialised before a thread starts, which is bad for software engineering.

Note 2: In official RMP ports, to include all floating-point processor configurations in one assembly file, floating-point operations are always encoded as machine code. This avoids assembler errors when the floating-point unit is absent; the correct save/restore routine is selected according to the configuration, so an undefined-instruction exception is not caused.

Note 3: On RMP, the recommended way to use a coprocessor is to let only one thread use it, so that its context need not be saved or restored, improving efficiency and real-time behaviour. For a very few special architectures whose coprocessor context is fully controlled by hardware and cannot be saved or restored by software,[26] the system must be guaranteed to have only one thread using that coprocessor.

### 7.1.4 Implementation of thread memory protection

Microcontrollers usually implement memory protection with a memory protection unit (MPU). Unlike a memory management unit (MMU), which performs address translation, an MPU only restricts access permissions to physical memory and cannot translate addresses. An MPU consists of a set of registers filled with the access attributes of protected physical memory regions. Each thread can have its own set of MPU register settings, thereby restricting its own memory accesses.

[25] For example the ARMv7-M RVM port.
[26] For example the DO-instruction-related registers of the DSPIC family.

The recommended implementation treats the MPU as a coprocessor and saves and restores its context in `RMP_CTX_SAVE` and `RMP_CTX_LOAD`. Note that each thread's MPU configuration must allow reading and writing RMP kernel memory and executing RMP code; this is because RMP has no true user mode and kernel mode, and all system calls are implemented directly as function calls. For this reason, memory protection implemented on RMP can only enhance the system's functional safety,[27] but cannot provide any information security.

### 7.1.5 Low-power design considerations

In low-power design, a tickless kernel is recommended, and an instruction that puts the processor into a sleep state should be inserted in the `RMP_Init_Idle` hook.[28]

## 7.2 Known factors that affect real-time performance in RMP

There are only two known factors that affect RMP's real-time performance, listed below. Both are easy to avoid in practical applications; generally, if the system's real-time performance is damaged by these factors, the application is not well designed.

### 7.2.1 The delay queue

RMP uses one delay queue to manage all delays. In this queue, thread timeouts increase from front to back. Thus, on each timer interrupt, RMP only needs to check the first thread in the queue. However, this design requires that, when a thread is inserted, the queue be traversed to find the correct insertion position so that timeouts remain increasing. The worst-case time complexity of this traversal is $O(n)$ in the number of threads already in the queue.[29] It can be shown mathematically that no data structure can make all timer operations $O(1)$, so applications must not use delay waits or delay blocking in very large numbers.

In addition, the delays provided by RMP are always system-wide, and a low-priority thread's delay operation also needs the scheduler lock, which may interfere with high-priority threads. If this causes problems, it is recommended that the user implement software-timer functionality with a small number of threads, each serving the delays of a different priority.

### 7.2.2 Deleting kernel objects

Some kernel objects, such as thread mailboxes and semaphores, can have a wait queue, and deleting such an object requires removing every thread in the wait queue one by one. If too many threads are waiting on an object, deleting it can cause the scheduler lock to be held for too long.

### 7.2.3 Locking the scheduler and disabling interrupts

RMP provides interfaces for locking the scheduler and disabling interrupts. However, these obviously harm the system's real-time performance, so they should be avoided as far as possible in applications. For example, if it is known that only one thread operates on a memory pool, the scheduler lock need not be used when allocating and freeing memory.

## 7.3 Reducing the interrupt latency of RMP

To protect critical sections from intrusion, RMP needs to lock the scheduler for certain operations, and the simplest implementation is to disable all interrupts. However, this makes the whole system lose im-

[27] Also called dependability; for example detecting common accidents such as stack overflow.
[28] On ARM, the WFI and WFE instructions achieve this; on MSP430, some bits of the STATUS register must be changed.
[29] An $O(\log n)$ data structure exists, but RMP's application scenarios generally do not need such a heavyweight implementation.

mediate response to external events; even transparent interrupts[30] unrelated to the system are disabled, increasing the delay from the stimulus source to execution of the interrupt vector.[31] In applications such as real-time electromechanical equipment or interface emulation, this behaviour is unacceptable. RMP therefore provides a series of methods to reduce or completely eliminate interrupt latency caused by critical sections. Whichever method is used, system interrupts[32] must use the context-saving macro provided by RMP, while transparent interrupts need only save and restore the registers they modify; the preemption priority of system interrupts must be set to the same value so that they cannot nest with one another.

### 7.3.1 Masking system interrupts by priority

If the architecture[33] provides the ability to mask interrupts below a certain priority, and the priorities of all system interrupts are below that value, masking interrupts at or below that value can disable only system interrupts without disabling transparent interrupts. On such architectures, official RMP ports always use masking instead of disabling all interrupts, so no interrupt latency is introduced. The mask level in the configuration header can be adjusted by the user.

### 7.3.2 Masking system interrupts by source

On architectures that cannot mask by priority, the same effect as priority masking can be achieved by disabling each system interrupt source in turn. However, because it is impossible to predict which system interrupt sources the user will use, RMP cannot provide a unified solution: the `RMP_INT_MASK()` and `RMP_INT_UNMASK()` macros in official ports operate on global interrupts, and `RMP_YIELD` also operates on global interrupts on entry and exit.

To achieve source-based masking, the official port must be modified as follows:

1. define `RMP_INT_MASK()` and `RMP_INT_UNMASK()` as the masking and unmasking operations for each system interrupt source;
2. after entering an interrupt, save the interrupt return address[34] and immediately enable global interrupts. This only requires inserting an instruction that enables global interrupts before calling RMP's official context-save macro, so the global interrupt disable lasts at most one instruction. If the hardware does not provide a way to set all system interrupts to the same preemption priority, the provided macros must be modified to mask each system interrupt source before enabling global interrupts, and before returning from the interrupt (1) disable global interrupts, (2) unmask the system interrupt sources, and (3) atomically enable global interrupts and return from the interrupt;[35] and
3. add masking and unmasking of system interrupt sources at the beginning and end of `RMP_YIELD`.

### 7.3.3 Delegating operations to a software interrupt

When there are many system interrupt sources or the interrupt controller is inconvenient to configure, masking and unmasking each system interrupt source creates a huge porting and execution burden. In this case, if the processor has a pendable software interrupt source or a spare software-triggerable peripheral interrupt source,[36] a higher-priority interrupt can delegate some operations to it. The software

[30] Transparent interrupts are those that do not call the `_ISR` series of functions and are unrelated to the operating system.

[31] The interrupt latency here is hardware interrupt latency, which is different from RTOS interrupt latency; the latter generally means the delay from sending in an interrupt to activation of the corresponding receiving thread, is an inherent value of the RTOS, and cannot be reduced.

[32] System interrupts call the `_ISR` series of functions and access operating-system critical sections, including the clock interrupt and the dedicated context-switch interrupt.

[33] For example the Cortex-M series, C28x, and similar architectures.

[34] Most architectures automatically push the return address on interrupt entry; for example RISC-V must push it manually before enabling interrupts, otherwise a transparent interrupt can overwrite it before it is saved and the system interrupt cannot return to its original context.

[35] The interrupt-return instruction can achieve this.

[36] Peripheral interrupt sources are generally always available and can be used for this purpose.

interrupt is then a system interrupt, and the higher-priority interrupt is a transparent interrupt, greatly reducing the number of system interrupt sources. If interrupts occur frequently, this method also reduces processor load, because a transparent interrupt only needs to save and restore the registers it uses.[37] However, for the delegated operation, both its own interrupt latency and its latency to the thread become longer, so this method has advantages and disadvantages.

To delegate operations, the official port must be modified as follows:

1. using the method of Section 7.3.2, add the software interrupt source, together with all system interrupts, to the source-masking objects;
2. in the transparent interrupt, record the system operation to be performed in a data structure and trigger the software interrupt source; and
3. in the software interrupt, read the system operation from the data structure and perform it on behalf of the transparent interrupt.

The design of the data structure is difficult because it may be modified by both the transparent interrupt and the software interrupt. If the data structure is a simple counter, the software interrupt can keep a local copy and compute the difference between the local copy and the current counter value to decide how many operations to perform; the counter can be read atomically by reading repeatedly until the value is stable. If the data structure must record complex additional information, it can be accessed with atomic instructions, or even with a lock-free or wait-free data structure; on processors without such instructions, the same effect can be achieved by briefly disabling the corresponding transparent interrupt. For the design of mutual-exclusion schemes, refer to the relevant literature; they are not described further here.

### *7.3.4 Delegating operations to the timer interrupt*

When a software-pendable interrupt source cannot be found or is inconvenient to use, but operation delegation is still desired, operations can be delegated to the timer interrupt. Its principle is similar to delegation to a separate software interrupt, except that (1) the timer interrupt handles the delegated operations, (2) the timer interrupt is triggered by the timer hardware rather than by a transparent interrupt, and (3) the system must be a ticked kernel.

To implement timer delegation, the official port must be modified as follows:

1. handle the timer interrupt, the only system interrupt source, using the method of Section 7.3.3;
2. in the transparent interrupt, record the system operation to be performed in a data structure; and
3. in the system tick hook `RMP_TIM_HOOK`, read the system operation from the data structure and perform it on behalf of the transparent interrupt.

This method effectively pushes all interrupt operations within a time slice to the timer interrupt, so these operations may be delayed for a long time; it can be used only when the thread does not need to know immediately that the interrupt occurred. If these operations are notifications rather than counts, multiple operations can be merged into one, which can produce considerable performance improvement.[38]

[37] On architectures with multiple register banks, a transparent interrupt can use another register bank to avoid saving and restoring entirely.

[38] This is especially true for devices with deep buffers, such as network cards and serial ports, where processing can wait until a certain amount of data has been received.

### 7.3.5 Limiting interference from non-critical interrupts on critical interrupts

In mixed-criticality systems, some transparent interrupt sources are not critical but have high priority.[39] If their interrupt frequency is not limited, they will inevitably consume a great deal of CPU and may even interfere with the execution of other interrupts that are critical but relatively less urgent, causing their interrupt latency to grow without bound.

To limit the frequency of these transparent interrupt sources, the following measures are needed:

1. prepare a counter for each transparent interrupt source; atomically decrement it on each interrupt, and when it reaches zero, disable the interrupt source; and
2. whenever the timer interrupt or a critical interrupt occurs, atomically (1) assign an appropriate initial value to the counter or (2) add a fixed value until it reaches the upper limit, and enable the corresponding interrupt source.

Each transparent interrupt source is then effectively confined to a deferrable server or sporadic server: it can be triggered only a limited number of times in a period, and excess triggers are lost without response. In this way, the interference suffered by a low-priority but high-criticality interrupt is bounded.

## 7.4 Reducing the memory footprint of RMP

When memory is especially precious, RMP must be configured carefully to save RAM and ROM. This situation is common on 16-bit microcontrollers such as MSP430 and low-end 32-bit microcontrollers such as Cortex-M0. In practice, the following methods can reduce RMP's memory footprint.

### 7.4.1 Reducing the number of preemption priorities

Unlike other operating systems, RMP's thread mailbox is built into the thread and costs nothing, while other functions consume no memory when unused. Therefore, only one compile-time option determines the kernel's RAM usage: the number of priorities supported by RMP.[40] Because RMP manages each priority with a doubly linked circular list, each priority consumes two machine words of RAM. On some tiny systems it is completely unnecessary to configure many priorities, so the number of supported priorities can be reduced to 3 to save the maximum amount of RAM. In this case, RMP's own RAM consumption can be squeezed below 128 bytes.[41]

### 7.4.2 Adjusting the vendor-provided library

Some vendor HAL libraries and startup files[42] statically allocate heap and stack. In an RMP system, the vendor-allocated heap is completely unnecessary and can be set to zero; the vendor-allocated stack is used by the OS kernel when responding to interrupts,[43] so it is only necessary to confirm the maximum stack consumption of the interrupt handlers and set the vendor stack size slightly above that number.[44] In addition, unused files in the vendor library need not be added to the project, which also helps reduce RAM and ROM usage. If the vendor library's memory usage is unacceptable, the peripheral registers can be operated directly.

[39] Urgent but not important, where occasional lost interrupts do not matter, such as remote-control decoding and streaming media transmission.

[40] That is, `RMP_PREEMPT_PRIO_NUM`.

[41] If the kernel interrupt-response stack is not counted, the actual size is 64 bytes.

[42] For example at the very beginning of the assembly startup file of a Cortex-M series microcontroller.

[43] That is, used as the kernel stack.

[44] A few dozen extra bytes are generally recommended.

### 7.4.3 Adjusting compiler options

Adjusting compiler options can also reduce ROM and RAM usage. Optimising for code size is especially effective for reducing ROM and sometimes also helps RAM.

### 7.4.4 Not using dynamic memory management

RMP has a built-in dynamic memory allocator. This allocator is not recommended on small-memory devices, because the allocator itself also has a fixed memory cost. On small-memory devices, all kernel objects and variables are recommended to be statically allocated.

## 7.5 Recommended rules for porting

On some architectures, porting RMP may present a dilemma. The following sections describe why these choices are difficult and give recommended handling rules.

### 7.5.1 Recommended rules for processor word length

On 32-bit or higher processors, the pointer length is generally one machine word, because these processors can generally use any register for indirect addressing and a machine word can address the whole address space; in this case `rmp_ptr_t` can be defined directly as one machine word. However, for processors with shorter words, (1) data and code space may be separate,[45] (2) the instruction-set design imposes great restrictions, and (3) the machine word may not address the whole address space; deciding the length of `rmp_ptr_t` therefore requires some skill.

#### 7.5.1.1 The case of 8-bit processors

On the vast majority of 8-bit processors, RMP cannot or need not be ported, because (1) their memory resources are too small, (2) to pursue memory utilisation efficiency, compilers use static overlay of local variables,[46] (3) to pursue silicon-area efficiency, the function call stack is implemented in hardware, and (4) applications are relatively simple and can be expressed with an RMS state machine. On the few 8-bit processors with abundant resources where RMP can be ported, almost all compilers default to a pointer length of 16 bits, because 8 bits can address only 256 bytes, which is too little for any application. Therefore `rmp_ptr_t` must be defined as at least 16 bits.

A very few 8-bit processors can even address a 24-bit memory space; two cases must then be discussed. If the internal binary representation[47] of either the data pointer or the function pointer is wider than 16 bits, then `rmp_ptr_t` must be defined as 32 bits on that architecture, which wastes space but is unavoidable. If the internal representation of both pointers is 16 bits, then `rmp_ptr_t` can be defined as 16 bits. The latter case is possible because there are two ways to address memory beyond 16 bits: (1) a segment register independent of the pointer, in which case, if the compiler's pointer does not include the segment register, the pointer is 16 bits[48] and the segment register must be changed manually by the user program; or (2) the toolchain guarantees that the variables and code used by the kernel are linked into the first 64 KiB,[49] while the rest of the address space is available to variables and code that are mostly not referenced by pointers. For details of segment registers and memory models, see the next two subsections.

#### 7.5.1.2 The case of 16-bit processors

Most 16-bit processors can be ported. They can also be divided into those that can address only a 16-bit address space and those that can address more. For the former, defining `rmp_ptr_t` as 16 bits is sufficient;

[45] Some architectures have separate data and code memory; data and code memory may also each have several subcategories.
[46] This may also be because the architecture lacks SP-relative addressing.
[47] For example some RL78 and similar architectures use a 20-bit pointer literal but a 16-bit internal binary representation.
[48] For example DSPIC and similar architectures.
[49] For example the AVR architecture.

for the latter, handle it like the 8-bit case according to how the higher memory space is addressed.

### 7.5.2 Rules for saving and restoring segment registers

Some processors have segment registers,[50] and some compilers may use a global variable pointer register under certain conditions.[51] Depending on how the compiler uses it, the segment register context can be handled differently.

#### 7.5.2.1 The segment register is set only by startup code

This is the easiest case. The segment register is set once at startup and never changed afterwards; this may be because the program is small enough to fit in one segment. If the program's behaviour can be confirmed to be like this, the segment register need not be saved or restored at all.

#### 7.5.2.2 The segment register is changed temporarily before each memory access

This is a relatively easy case. The compiler makes no assumption about the segment register's value and points it at the required segment temporarily before a memory access. Because the segment register value may change and an interrupt may occur between the change and the use, the segment register must be saved and restored.

#### 7.5.2.3 The segment register must hold its initial value before a function call

This is the most difficult case. The compiler not only generates code that modifies the segment register inside a function, but also assumes that the segment register has its initial value before entering every function, including an interrupt vector; this is generally done to optimise program efficiency. If startup code can point the segment register at the most frequently accessed segment at initialisation, segment setup can be omitted between function calls, reducing code size. Functions that access other segments must save and restore the segment register at their entry and exit.

In this case, the segment register must be saved and restored in the context, and its initial values must be saved in `_RMP_Start`; if an interrupt vector calls a C function, the segment registers must be set to the values saved by `_RMP_Start` before the call.[52] The advantage of this approach is that, regardless of how the toolchain and startup code set these values, they are necessarily captured by `_RMP_Start`, so there is no need to obtain them from a variety of linker scripts. Of course, if the toolchain is an open standard product such as GCC, the values can also be obtained directly from the linker script.

The segment-register usage style of some toolchains changes with the selected memory model. Therefore, on architectures with segment registers, almost all official RMP ports use the above method to eliminate the risk of segment errors, even when it may be unnecessary. There is only one exception: the architecture plans to support only one toolchain, and that toolchain documents fixed initial values for the segment registers.[53]

### 7.5.3 Recommended rules for memory models

On some 8-bit and 16-bit processors, the presence of segment registers allows the program to use one or more code segments (CODE), data segments (DATA), or stack segments (STACK), producing different memory models. When the program uses one segment, its pointers are 16 bits but the address range is limited to one segment; when the program uses multiple segments, its pointers are 24 or 32 bits and the address range covers the whole memory space. Some architectures even divide the same memory into several classes by address space, each with a different pointer length. In addition, the default pointer types of the code, data, and stack segments can be set independently: if the code segment has C pointer

[50] For example DSPIC and similar architectures.
[51] The only exceptions are MIPS and RISC-V.
[52] Also in `RMP_YIELD`; if a dedicated thread context-switch interrupt vector is not used.
[53] For example the AVR architecture, for which only a GCC port is provided.

lengths, the data segment has D pointer lengths, and the stack segment has S pointer lengths, there are C × D × S memory models. Some architectures' segment-register schemes are even more varied, because they do not provide architectural segment registers but require the user to implement memory-mapping circuits externally with discrete logic and I/O ports.[54]

For simplicity, RMP cannot provide a port for every case, and RMP assumes that all pointers have the same length, which requires the default memory models of the code, data, and stack segments to be the same. This can create a dilemma when porting: if the largest memory model is always chosen without thought, `rmp_ptr_t` is 32 bits, greatly increasing the processor's burden, especially when the program is small; if the smallest memory model is always chosen, `rmp_ptr_t` is 16 bits, the processor burden is lighter, but a larger address space cannot be accessed. For this reason, processors are divided into the following three classes, with a recommended scheme for each.

#### 7.5.3.1 The common memory model is the large model

A small number of processors[55] have used the large memory model since their introduction, and most of their toolchains default to `sizeof(void*)` = 4. This is because they have too many memory classes and some classes have too small an address range; even the toolchain authors find the many combinations difficult to handle and use long pointers to avoid the complexity.

On these processors, official RMP ports use the large model: `rmp_ptr_t` is defined as 32 bits, the upper 16 bits are the segment register value, and the segment register and the raw pointer are saved and restored together.

#### 7.5.3.2 The common memory model is the small model

Most processors use the large memory model only when the user explicitly requests it; most of their toolchains default to `sizeof(void*)` = 2, to maximise efficiency in performance-sensitive parts of the program. When the program needs to access more memory, the programmer declares variables and functions as far symbols with the `far` keyword, and the compiler automatically generates the corresponding segment and cross-segment access code, so that the wider memory address space can be accessed even in the small model.

On these processors, official RMP ports use the small model with segment register save/restore: `rmp_ptr_t` is defined as 16 bits, but the segment register is switched during a context switch to support thread access to other segments. The user must ensure that all kernel data structures[56] are in the same segment as the kernel data segment, and that all thread entry points[57] are in the same segment as the kernel code segment; however, once a thread starts, it can access variables in other segments and jump to other segments to run without this restriction.

When porting, the stack segment must be handled specially depending on the case: (1) if the processor does not provide a stack segment register, or the common systems it forms have RAM close to 64 KiB,[58] the stack segment needs no special handling; (2) if the processor provides a segment register and its RAM is commonly much larger than 64 KiB,[59] then `RMP_STKSEG_ENABLE` can be defined as 1 to store the stack segment register in the thread control block. Once this macro is enabled, the formats of the "create thread" system call and the "thread stack initialisation" low-level call both change, and an additional `Segment` parameter appears before the `Stack` parameter so that the user can pass the stack segment register value.

Note: In case (2), the stack segment register cannot be saved on the stack, because restoring the stack segment register from the stack requires popping, and popping itself uses the stack segment register, creating a paradox.

[54] For example the memory mapper of the 6502-based Famicom/NES.
[55] For example the 8051 and 80251 architectures.
[56] Including all object control blocks and the messages in a message queue.
[57] Do not declare them as far.
[58] Common on microcontrollers: they either have no stack segment register or have less than 64 KiB of RAM, so the stack segment register is always fixed.
[59] Common on general-purpose microcomputer systems such as the 8086.

#### 7.5.3.3 The common memory model cannot be inferred

Some processors[60] have a memory model that cannot be inferred; most of their toolchains default to `sizeof(void*)` = 2, but allow the user to write custom segment-switching code to work with external custom circuits for cross-segment access.

For such architectures, official RMP ports always use the small model and ignore the segment registers. The user must make a small custom modification to the low-level port to adapt it to the hardware structure.

### *7.5.4 Recommended rules for porting compatibility*

Almost all microcontroller vendors combine a base instruction set with a series of optional coprocessors[61] to serve different market segments. Some vendors even increase the register length of an older architecture[62] to meet new requirements. When porting RMP, a simple approach is to write a separate port for each coprocessor combination, but this leads to a huge amount of duplicated code between ports. We therefore define some rules to decide which coprocessor combinations should be merged into one port.

#### 7.5.4.1 The link-and-run rule

This is the most important rule for deciding whether to merge ports. If there is a subset of the binaries for several coprocessor combinations[63] that can be linked for any combination and the resulting final binary can run on that combination, those combinations can be merged into one port. In this case, an additional coprocessor-selection macro `RMP_XXX_COP_YYY` is added to each chip configuration header, where XXX is the port name and YYY is the extension instruction set; selecting the macro selects the coprocessor combination.

The compatibility between coprocessors can be classified as "contains", "and", "or", "not", and "xor": for two coprocessors A and B, "contains" means that if A exists then B must exist, but the converse need not hold; "and" means that if A exists then B must exist and vice versa; "or" means that the existence of A and B is independent and neither affects the other; "not" means that only one of A and B may exist, but neither need exist; "xor" means that exactly one and only one of A and B must exist. The compatibility between combinations is confirmed at compile time by the port's low-level code, and an error is reported through `#error` when the user selects an invalid combination. Generally, microcontroller coprocessors are simple and do not involve very complex compatibility judgments.

Note: The link-and-run rule concerns not only the common instruction subset of the coprocessor combinations but also how mainstream compilers handle each combination. In the following cases, even if a common instruction subset exists, the software may still be incompatible: (1) the common subset is not Turing-complete and cannot form an application on its own; (2) the common subset cannot be selected alone in the compiler, that is, a binary containing only that subset cannot be generated; (3) the calling conventions of the combinations are completely different, so that even two arguments cannot be passed in a unified way;[64] (4) the memory models of the combinations are completely different and incompatible; and (5) the stack alignment required by the combinations is different. If cases (1)–(4) occur, the rule is not satisfied[65] and a separate port must be provided for each combination; if only case (5) occurs, the rule is satisfied, and the stack alignment of the merged port is the maximum of all combinations.[66]

[60] For example 6502 and Z80 architectures, but not Rabbit 2000, which officially defines segment registers.
[61] Or extension instruction sets.
[62] For example the MSP430X architecture.
[63] Including the no-coprocessor case.
[64] Assembly may be possible, but its development efficiency is too low to be meaningful.
[65] For example ARMv6-M and ARMv7-M are treated as two independent architectures because cases (3) and (4) occur.
[66] For example C64x, C67x, C64x+, C67x+, and C66x are merged as the C6x architecture because only case (5) occurs.

#### 7.5.4.2 The mental-model rule

This is the secondary rule for merging ports. If either of the following two conditions holds, the coprocessor combinations are considered to have a unified mental model and can be merged into one port: (1) the vendor explicitly states in its official documentation that the combinations are derivatives of the same processor;[67] or (2) although the vendor does not state this explicitly, its technical manuals often discuss the combinations together in a "bound volume".[68]

For coprocessor combinations that satisfy the link-and-run rule but not the mental-model rule,[69] official RMP ports always keep them separate to preserve the independence of their mental models.

#### 7.5.4.3 The high-repetition rule

This is the least important rule for merging ports. If the above two rules both hold and the low-level code of the combinations has a considerable degree of repetition, the combinations can be merged into one port: high repetition indicates that (1) the coprocessor combinations are very similar and (2) the products corresponding to these combinations occupy the same or similar market niche.

For coprocessor combinations that satisfy the link-and-run and mental-model rules but not the high-repetition rule, official RMP ports always merge the C code into one file but may provide separate assembly files. In this case, an additional coprocessor-selection macro `RMP_XXX_COP_YYY` is also added to each chip configuration header.

## 7.6 Vendor library compatibility

With developer time extremely valuable, most microcontroller vendors provide firmware libraries. Accessing on-chip peripherals through these easy-to-learn library functions is much easier than reading and writing registers. Unfortunately, some firmware libraries' low-level functions may conflict with the operating system and need to be modified.[70] Common conflicts occur in interrupts and delays; their solutions are described below.

### *7.6.1 Interrupt enable/disable compatibility*

When operating some peripherals, the read/write sequence must be completed within a specified time without interruption. Firmware libraries therefore often disable interrupts to complete the operation and then enable them again. Do not call such library functions while the system is holding the scheduler lock; otherwise they will enable interrupts before the scheduler is unlocked, without notifying the operating system, and crash the system. If it is uncertain whether these library functions may be called while the scheduler is locked, consider replacing their interrupt enable/disable functions with RMP's scheduler lock/unlock functions, and define `RMP_INT_MASK` and `RMP_INT_UNMASK` to mask all interrupts.

### *7.6.2 Delay-function compatibility*

When operating some peripherals, the read/write sequence often requires waiting for the peripheral to become ready, so delay functions are inserted between steps. Firmware libraries implement delay functions in many ways: busy loops, timer polling, or timer interrupts. Some libraries even use several implementations simultaneously for different purposes.

A busy-loop delay function delays by running a useless instruction sequence. Its behaviour is the same as ordinary code and it is compatible with RMP without modification. It can also be used with the

[67] For example ARMv7-M FPV4_SP, FPV5_SP, and FPV5_DP.
[68] For example the PIC24, PIC33, and various other DSPIC derivatives.
[69] For example ARMv6-M and ARMv7-M.
[70] Some vendors have considered this and added a "weak symbol" attribute to the default implementation; in that case they can simply be overridden.

scheduler locked or interrupts disabled, because it does not depend on the interrupt system. Of course, if it can be confirmed that the library never calls this delay function while the scheduler is locked or interrupts are disabled, it can be rewritten to use `RMP_Thd_Delay`, which saves processor performance.

A timer-polling delay function delays by initialising a timer and repeatedly polling it until the delay is complete. It is more accurate than a busy loop and can also be used with the scheduler locked, but it may interfere with RMP's tick timer. The delay function may modify (1) the timer's counter value, (2) the timer's upper count value, or (3) the timer's operating frequency, any of which may cause the system to malfunction. Worse, it may disable the timer interrupt during initialisation, causing RMP's scheduler to lose control. When this happens, the function can be rewritten to (1) use another timer, (2) use a busy loop, or (3) call `RMP_Thd_Delay`. Method (3) is the most processor-efficient, but it is not always possible, because the library may call the delay while the scheduler is locked or interrupts are disabled; if no other timer is available at that time, the busy-loop method provided by `RMP_Thd_Loop` must be used.

A timer-interrupt delay function initialises a timer interrupt, increments a global variable in the interrupt, and repeatedly polls the global variable during the delay. It has high delay accuracy and a wide delay range, and can also provide a timestamp variable for querying, but it may interfere with RMP's tick timer. However, this delay function can only be used with interrupts enabled, so it can be simply replaced with `RMP_Thd_Delay`. If the global variable it produces is also read, it can be replaced with `RMP_Timestamp`.

### 7.6.3 Code-generator compatibility

To make microcontrollers easier to use, some vendors provide code generators on top of their library functions. The output of these generators may conflict with RMP. Three common conflicts are:

1. **Generated code containing a `main` function.** To be compatible with vendor startup code that calls `main`, RMP declares `main` inside the kernel. When generator-generated framework code is compiled with RMP, `main` is defined twice and a link error occurs. Because these `main` functions mostly contain simple hardware initialisation code, the `main` function can be renamed or macro-defined to `hw_main`, and `RMP_LOWLVL_INIT` in the configuration header can be defined as it. To simplify the system startup mental model, RMP does not provide an independent `RME_Kmain` as RME does, so other code must stay out of the way of `main`.[71]
2. **Generated code that initialises the watchdog or enables unrelated peripheral interrupts, which may unexpectedly interfere with the system.** The best solution is to configure the generator not to generate such code; if this is impossible, the generated code must be modified manually to remove these interference items, or additional operations[72] must be added during system startup and operation so that they do not cause interference.

## 7.7 RMP's deviations from the C standard and MISRA

### 7.7.1 C undefined behaviour: type-punning pointer aliasing may exist and does not follow strict aliasing

Type-punning pointer aliasing is a widely used practice in C that allows two pointers of different types to refer to the same block of memory. It is very common in network protocol stacks, where packets must be assembled or disassembled and byte order changed, and in operating systems, where unsigned integer pointers and structure pointers are converted back and forth to operate on objects. Unfortunately, C's

[71] After entering `main`, some memory controllers, segment registers, and other hardware may not yet be initialised, and execution is not in a thread context; certain kernel operations at this point may fail for hard-to-diagnose reasons, while doing them after everything is initialised avoids such problems. In addition, we want the system entry to always be `main`, so that its stack can be used directly to initialise the interrupt-response stack, which suits bare-metal toolchains; for these two reasons, we do not use the independent kernel entry function design common in other RTOSes.

[72] For example feeding the watchdog.

strict aliasing rule does not allow two pointers of different types to refer to the same memory unless the types are compatible or one is converted to `char`;[73] there is also no general way around this. There are generally three ways to avoid type punning:

1. use a `union` to convert between types. Unfortunately, this is defined behaviour in C but not in C++, and RMP code may need to be compiled as C++ because it may link directly with libraries written in C++,[74] so it may still be undefined behaviour. Moreover, this method cannot handle array objects whose length is not known at compile time unless they are converted one by one at a performance cost. Finally, using `union` this way does not conform to the design spirit of `union` and abuses the language rules.
2. use `char*` for the operation, because `char*` can always alias any pointer. Unfortunately, existing compilers optimise this very poorly; some quite good compilers even compile these operations into byte-by-byte access instructions, especially low-quality embedded compilers. This is therefore a poor solution.
3. use `memcpy` to copy content between two incompatible pointers, and copy it back after the operation, hoping that the compiler can detect this and optimise away the duplicate `memcpy`. This cleverly exploits the fact that `memcpy` takes `char*` arguments. Its immediate problem is that some low-quality embedded compilers may not be smart enough to optimise `memcpy` away, causing the generated code to actually call `memcpy` twice and incur a huge performance loss. Once `memcpy` is not optimised away, a local copy buffer declared inside the function may be allocated on the stack, greatly increasing stack consumption and causing stack overflow. Finally, this use of `memcpy` does not conform to its design intent and abuses a language loophole; the C standard does not encourage this practice.

In view of the problems with these three methods, RMP does not use them; it uses type-punning pointer aliasing directly and disables the relevant optimisation at compile time.[75]

### 7.7.2 C undefined behaviour: a byte is always eight bits and a `char` is always one byte

C does not define the length of `char`; it only says that `char` must equal the width of a single character in the system's basic character set.[76] In theory, a `char` can be 16 or 32 bits, or even more. On all current mainstream compilers and architectures, `char` is always 8 bits, so the code makes this assumption and avoids an unnecessary "char length" configuration macro.

In fact, RMP does not use bits above bit 7 of a `char`, and does not operate on bytes or measure longer data types in bytes,[77] so it can still work on a platform where `char` is wider than 8 bits.[78] However, such a port must be careful: (1) `rmp_u8_t` is no longer literally 8 bits but the smallest integer type that can express 8 bits,[79] (2) all addresses in this manual are word-addressed, and (3) all data-structure sizes and memory lengths in this manual are in words, not bytes.

### 7.7.3 C undefined behaviour: a pointer's actual representation is always a machine-recognisable address

C does not define how a pointer should be represented in memory. In theory, a pointer variable's actual representation can be anything, as long as dereferencing it returns the value of the variable it points to

[73] Referring to C90; the same applies below.
[74] If `extern "C"` is not desired.
[75] For example GCC's `-fno-strict-aliasing` option.
[76] Referring to C90; the same applies below.
[77] It uses bit operations and measures by `char`, which works on all processors.
[78] Some early digital signal processors, such as the C28x compiler, do this, but modern digital signal processors rarely do so; see the C28x port.
[79] Similar to `uint_least8_t` in `stdint.h`.

according to C semantics.[80] This makes it impossible, under strict C, to align a pointer to a memory granularity by masking some of its bits with a bitwise AND. We therefore assume that a pointer's actual representation is always a machine-recognisable binary address, which is true on all architectures that RMP supports or plans to support.

### 7.7.4 C undefined behaviour: converting a pointer to an unsigned integer no smaller than a machine word always preserves its actual representation

C does not specify how a pointer's actual representation should change when it is converted to an unsigned integer. It only specifies that if a pointer is converted to a sufficiently large unsigned integer and then converted back, the resulting pointer must compare equal to the original. If the representation changed when converted to an unsigned integer, we would face a problem similar to that described in Section 7.7.3. We therefore assume that converting a pointer to an unsigned integer of no less than one machine word always preserves its actual representation, which is true on all architectures that RMP supports or plans to support.

In fact, RMP does not forbid adding anything to the high bits of a wider unsigned integer after conversion,[81] as long as those high bits are discarded when converting back to the original pointer type. RMP only masks the lowest few bits of the unsigned integer and then converts it back to (1) the original pointer type or (2) a pointer type with alignment requirements no stricter than the original pointer, pointing to the same memory space; it never involves the potential high-bit content. However, such a port must be aware that the unsigned integer obtained from a pointer may contain unexpected high-byte garbage.

### 7.7.5 C undefined behaviour: all "exactly fitting" structures are automatically packed

C does not define the memory alignment of variables inside a structure. In theory, the compiler can decide the alignment of structure members arbitrarily, as long as it guarantees the structure's access semantics.[82] It does not even forbid aligning a structure containing only one type to any larger granularity. Therefore, even if a structure is "clean" in terms of types, we cannot guarantee that its internal variables are packed. To solve this lack in the C specification, almost all compilers provide additional mechanisms for packing structure members,[83] but their methods are like the Tower of Babel: some compilers use `#pragma`, some use `__attribute__`, and the position of the mechanism in the structure declaration differs between compilers. Compilers that use `#pragma` are especially troublesome, because a macro cannot expand to a `#pragma`; the `#pragma` must be inserted directly into the code file.

To avoid these problems, RMP assumes that the compiler always automatically packs all "exactly fitting" structures, where "exactly fitting" is defined as follows:

1. the size of every "ultimate base type" in the structure is a power of two, where "ultimate base type" means the compiler's native data type obtained by recursively composing the structure type;
2. if the structure is packed, the offset of every variable of an ultimate base type is naturally aligned to the size of that type; and
3. the structure's base-address alignment requirement is no stricter than the alignment requirement of its longest ultimate base type.

[80] This does not even prohibit a difference between the representation before and after a round trip, as long as the two values always compare equal as pointers.

[81] For example the RL78 CC-RL compiler inserts `0x000F` in the high bits of an unsigned integer converted from a memory pointer, to indicate that it points to the memory space.

[82] For example, aligning every variable in a structure containing only `char` to 1 GiB; C semantics permits this absurd behaviour.

[83] The best known example is GCC's `__attribute__((packed))`.

Almost all compilers satisfy these requirements, because the only purpose of aligning variables is to prevent unaligned accesses that may affect performance or cause exceptions. RMP therefore does not guarantee support for compilers that do not satisfy them.

### 7.7.6 C undefined behaviour: external identifiers are extended to 31 characters

C90 specifies that only the first 6 characters of an external identifier are guaranteed to be significant. This is far too few for any operating system. RMP therefore extends the limit to 31 characters, the same as the limit for internal identifiers. All modern compilers support this.

### 7.7.7 MISRA-C deviations

Table 7-1. RMP deviations from the Required part of MISRA-C:2012.

| Rule | Explanation |
|---|---|
| 1.3 | Some of the undefined behaviour described above is necessary for low-level software such as an operating system. |
| 5.1 | RMP extends the significant length of external identifiers to 31 characters. This is necessary for the software engineering of an operating system. |
| 8.4 | RMP uses a special header-file writing method to avoid this problem and allows a variable's declaration and definition to be in one place. This improves software quality. |
| 11.3 | Type-punning pointer aliasing and pointer-to-integer conversion are necessary for software such as an operating system. |
| 11.6 | RMP in fact uses such aliasing only in (1) structures and sub-structures, (2) HAL interfaces, and (3) memory pools. |
| 11.9 | To avoid conflicts with other third-party source packages, RMP uses `RMP_NULL` rather than `NULL` for a null pointer. |
| 14.3 | RMP uses many compile-time boolean expressions in its configuration check. These are passed directly to `RMP_ASSERT` to ensure that RMP is configured correctly. This improves software quality. |
| 17.7 | Because RMP's HAL must define a uniform interface for all supported hardware, some HAL or debug-print return values are redundant on some architectures. |
| 21.1 | RMP declares the reserved macros `__HDR_DEF__`, `__HDR_STRUCT__`, and `__HDR_PUBLIC__` for its own use. This does not violate the rule against undefining other system-reserved macros. See also 20.5. |

Table 7-2. RMP deviations from the Advisory part of MISRA-C:2012.

| Rule | Explanation |
|---|---|
| 2.3, 2.5 | In the HAL, especially in firmware libraries, some macros and data types are supplied together. They may be unused in the current configuration but must be kept for future configuration adjustments, which helps maintain consistency and improve code quality. These macros are well commented and documented. |
| 2.7, 8.7 | Because RMP's HAL must define a uniform interface for all supported hardware, some HAL function parameters are redundant on some architectures, and some functions declared `extern` are not actually called. |
| 8.9 | Some global variables in RMP are referenced by only one function but are not placed inside that function, to make code debugging easier for the user. |
| 11.4, 11.5 | Converting an unsigned number or `void*` to an ordinary pointer is necessary for software such as an operating system. See 11.3 for details. |
| 15.5 | Some RMP functions may have multiple exits. These exits are well commented, and other possible implementations would violate other requirements or recommendations and break the project's overall abstraction. |
| 20.5 | To ensure that every function and variable is declared only once (including its `extern` form) and to systematically prevent implicit-declaration errors, RMP internally uses a special segmented inclusion header format. This format requires `#undef`. The `#undef` is used only for this internal format, is not exposed outside the kernel, and is followed consistently throughout the system and documented in detail. |

RMP is strictly checked against all three classes of MISRA-C:2012 [2] rules, and all deviations are approved. RMP has no deviations from the Mandatory class. The deviations from the Required class are listed in Table 7-1, and the deviations from the Advisory class are listed in Table 7-2.

## 7.8 Chapter bibliography

[1] ISO, "Programming languages — C," International Organization for Standardization, ISO/IEC 9899:1990(E), 1990.

[2] MISRA Association, *MISRA C:2012: Guidelines for the Use of the C Language in Critical Systems*. MIRA, 2013.