
# Reference Manual

Author(s): D. van de Spijker
Version: R01
Date: 2018-06-21

# Document history

| Rel. | Date | Changes |
|---|---|---|
| R01 | 2018-06-21 | First version |

# Open issues

| Issue | Section | Description | Responsible | Due Date |
|---|---|---|---|---|
| 1. | ref | issue | - | due date |

# Contents

# List of Figures



# List of Tables



# List of Listings

# 1. Abbreviations and definitions

This chapter describes the used abbreviations and convention definitions throughout this document.

## 1.1. Abbreviations

| | |
|---|---|
| API | Application Programming Interface |
| CLI | Command Line Interface |
| ID | Identity |
| OS | Operating System |
| RTOS | Real-Time Operating System |
| UART | Universal Asynchronous Receiver Transmitter |
| RAM | Random Access Memory |
| ROM | Read-Only Memory |

## 1.2. Definitions

| | |
|---|---|
| '*a*' | Numeric binary notation (*a* can be multiple 0s or 1s). E.g. '`010`' is a 3-bit value representing the binary number two. This kind of notation implies a specific bit length. |
| '*aa.aaaa*' | Numeric binary notation with '.' separations for clear reading of long binary numbers. |
| 0x*a* | Numeric hexadecimal notation (*a* can be a digit 0 through 9, A through F). E.g. '`0x1A`' is hexadecimal number twenty-six. This kind of notation does not directly imply a bit length. |
| 0x*aa.aaaa* | Numeric hexadecimal notation with '.' separations for clear reading of long hexadecimal numbers. |
| *a*d | Numeric (explicit) decimal notation. This kind of notation does not directly imply a bit length. |
| X[b:a] | Vector notation for vector X with bit range b downto a (little endian notation). |
| *OsType_t* | Type definition notation |
| **SomeFunction** | Function definition notation |
| SOME_MACRO | Macro(expansion) notation |

# 2. Introduction

## 2.1. What is Prior RTOS?

Prior RTOS is an embedded Real-Time Operating System with a small footprint and high customizability enabling it to run smoothly on even the smallest devices such as Atmel's ATMega series. This combined with the fact that the kernel is entirely written in C99 makes it portable to almost any microcontroller or microprocessor architecture. With Prior RTOS the development of event driven real-time applications becomes a painless process using the wide variety of intuitive API functions.

## 2.2. Core principles

### Usability

The API functions are named intuitively, take few arguments, and have straightforward behavior. Furthermore, sharing an object, such as a mailbox between two tasks does not require a semaphore of any kind because all default modules implement an inherent locking mechanism.

### Integrity

Data integrity is an important pre-condition in order for a system to perform as required. Prior RTOS has built-in features to help meet this pre-condition:

- Resources referenced by ID preventing direct manipulation.
- Internal list verification to detect corruption.
- Ownership of resources avoids unauthorized access.
- Memory padding to detect overruns.

### Flexibility

The architecture design of Prior RTOS provides flexibility in the composition of the OS. Only a small set of modules is mandatory, additional modules can be enabled in the configuration. Furthermore, custom modules can be added and integrated with the rest of the kernel.

## 2.3. Features

- Cooperative scheduler based on weighted FIFO.
- Priority system with 4 categories to indicate scheduling/timing constraints and a priority level ranging from 1 through 5.
- Software Timer module with possible sub-millisecond intervals.
- Inter-Task Communication (ITC) system providing eventgroups, mailboxes, semaphores/mutexes and ring-buffers.
- Global Events allowing tasks to Wait and Poll for all possible events.
- Advanced Memory Management provides protected memory pools for dynamic allocations.
- Low OS Tick overhead: 750 clock-cycles (on AVR8).
- Small footprint kernel with a minimal size of 15 kB ROM and 350 Bytes of RAM (O0).
- Configurable memory usage, modules and utilities to tailor the OS to the application.
- Modular design allowing custom modules to be added.

# 3. Architecture

## 3.1. Overview

PriorRTOS consists of a number of parts:

- Port: The Hardware Abstraction Layer (HAL) for the OS. Discussed in Chapter 6.
- The kernel: All essential modules of the OS.
- Event System: Used throughout the OS to communicate events. Discussed in Chapter 7.
- Modules: Optional modules that extend the OS' functionality. Can be enabled and disabled in the configuration header.
- API: Application Programming Interface (API): Interface between the OS and the application.

A visual representation of these parts is shown in Figure 3.1.

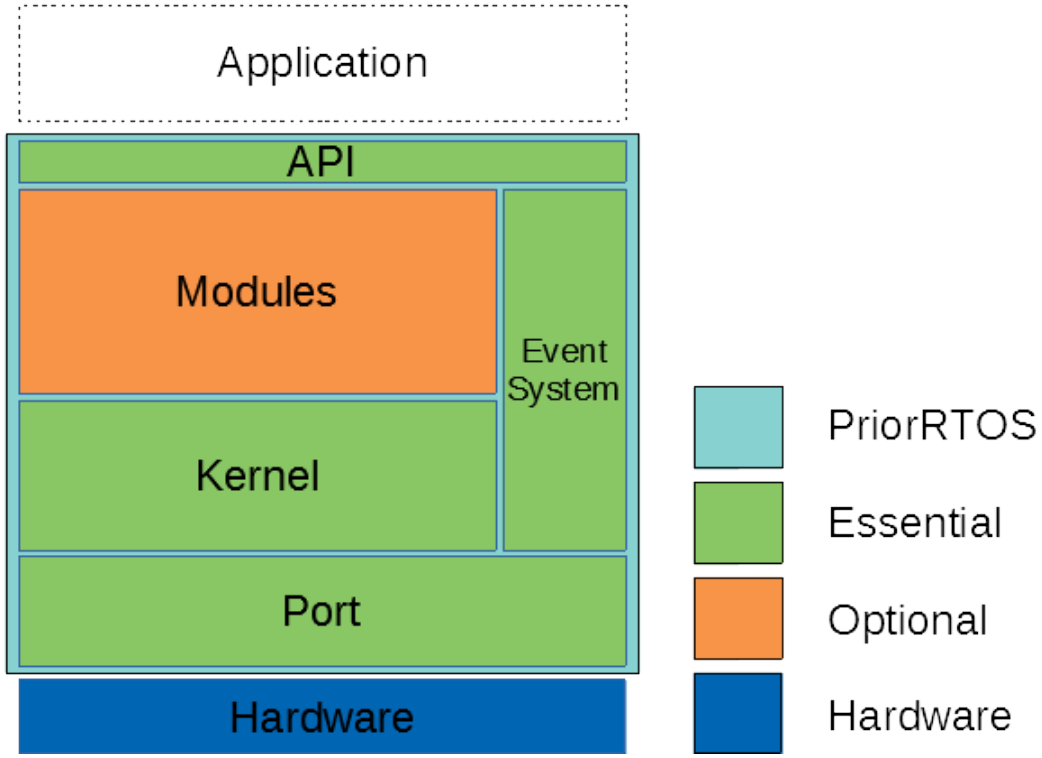


*Figure 3.1: Architecture*

## 3.2. Modules

Table 3.1 states the modules (essential and optional) that are currently available with their respective API.

*Table 3.1: Modules*

| Name | Essential/Optional | Category | Description | Reference |
|---|---|---|---|---|
| Event | Essential | Kernel | Event system | Chapter 7 |
| Core | Essential | Kernel | Scheduler and task switch logic | Chapter 9 |
| Task | Essential | Kernel | Task management | Chapter 10 |
| Memory | Essential | Kernel | Memory management | Chapter 11 |
| Timer | Optional | Timing | Software timers | Chapter 12 |
| Semaphore | Optional | *ITC | Semaphores and mutexes | Chapter 13 |
| Mailbox | Optional | ITC | Addressed data exchange | Chapter 14 |
| Eventgroup | Optional | ITC | Flags with coupled events | Chapter 15 |
| Ringbuffer | Optional | ITC | Circular FIFO buffer for streaming | Chapter 16 |
| Logger | Optional | Debugging | Logging using debug UART | Chapter 17 |
| Shell | Optional | Debugging | Command Line Interface | Chapter 18 |
| Convert | Optional | Utility | Conversion functions | Chapter 19 |

*ITC = Inter-Task Communication

# 4. Configuration

PriorRTOS uses a single file, *PriorRTOSConfig.h*, to manage its configuration. Information about the configuration macros is stated below.

- All configuration macros are prefixed with PRTOS_CONFIG_ .
- Settings requiring a specific value such as a frequency or interval are suffixed with their respective unit e.g. PRTOS_CONFIG_F_CPU_HZ or PRTOS_CONFIG_OS_HEAP_SIZE_BYTES.
- PRTOS_CONFIG_ENABLE_ settings expect a 1 to enable, and a 0 to disable.
- PRTOS_CONFIG_USE_ settings must be defined if used, and undefined/commented out if not used.

## 4.1. Configuration file

```
/************ General Settings ************/

/*******************************************************
 * @macro: PRTOS_CONFIG_F_CPU_HZ
 *
 * @desc: Target CPU frequency in Hertz, e.g. 48000000 =
 * 48MHz.
 *
 * @dtype:     U32_t
 * @unit:      Hertz.
 *******************************************************/
#define PRTOS_CONFIG_F_CPU_HZ                                   72000000


/*******************************************************
 * @macro: PRTOS_CONFIG_F_OS_TIMER_HZ
 *
 * @desc: Non-prescaled Clock frequency of the hardware
 * timer used to generate the OS tick interrupt.
 *
 * @dtype:     U32_t
 * @unit:      Hertz.
 *******************************************************/
#define PRTOS_CONFIG_F_OS_TIMER_HZ      PRTOS_CONFIG_F_CPU_HZ


/*******************************************************
 * @macro: PRTOS_CONFIG_F_OS_HZ
 *
 * @desc: The frequency of the OS tick interrupt in Hertz.
 * A higher frequency results in a more responsive system,
 * but also increases CPU load.
 *
 * @dtype:     U16_t
 * @unit:      Hertz.
 *******************************************************/
#define PRTOS_CONFIG_F_OS_HZ                                    1000


/*******************************************************
 * @macro: PRTOS_CONFIG_IRQ_PRIORITY_TYPE
 *
```

```
 * @desc: Defines the Interrupt Request priority type.
 ******************************************************/
#define PRTOS_CONFIG_IRQ_PRIORITY_TYPE                  U32_t

/*******************************************************
 * @macro: PRTOS_CONFIG_OS_TICK_IRQ_PRIORITY
 *
 * @desc: OS tick interrupt priority.
 *
 * @dtype:     IrqPriority_t
 ******************************************************/
#define PRTOS_CONFIG_OS_TICK_IRQ_PRIORITY               0


/*******************************************************
 * @macro: PRTOS_CONFIG_USE_SCHEDULER_COOP
 *
 * @desc: The kernel will use the cooperative scheduler.
 * Tasks must yield voluntarily (by calling TaskSuspendSelf)
 * in order for the scheduler to switch tasks.
 ******************************************************/
#define PRTOS_CONFIG_USE_SCHEDULER_COOP

/*******************************************************
 * @macro: PRTOS_CONFIG_USE_SCHEDULER_PREEM
 *
 * @desc: The kernel will use the pre-emptive scheduler.
 * A task can be switched out by the scheduler when
 * needed e.g. a higher priority task is ready. A task
 * can also call TaskSuspendSelf to suspend execution
 * or call TaskSuspend to suspend another task.
 ******************************************************/
/* !!!NOT YET AVAILABLE IN V 0.4.X!!! */
/* #define PRTOS_CONFIG_USE_SCHEDULER_PREEM */

/*******************************************************
 * @macro: PRTOS_CONFIG_USE_SYS_CALL_NO_BLOCK
 *
 * @desc: All system calls will be implicitly non-blocking
 * even when using the pre-emptive scheduler, meaning
 * that they will poll for events instead of waiting.
 ******************************************************/
#define PRTOS_CONFIG_USE_SYS_CALL_NO_BLOCK


/*******************************************************
 * @macro: PRTOS_CONFIG_EVENT_LIFE_TIME_TICKS
 *
 * @desc: Defines the amount of ticks every emitted event
 * lasts. A higher number of ticks enables tasks to respond
 * to events that occurred further in the past.
 *
 * @dtype: U8_t
 * @unit: ticks
 ******************************************************/
#define PRTOS_CONFIG_EVENT_LIFE_TIME_TICKS              3
```

```
/***** List settings. *****/

/*************************************************
 * @macro: PRTOS_CONFIG_EVENT_LIFE_TIME_TICKS
 *
 * @desc: Enable the verification of all Lists existing
 * within the kernel. After every access the list's
 * navigation is checked for any corruption.
 *************************************************/
#define PRTOS_CONFIG_ENABLE_LIST_INTEGRITY_VERIFICATION 1

/*************************************************
 * @macro: PRTOS_CONFIG_USE_SORTED_LISTS
 *
 * @desc: The List API will use sorted lists where
 * possible. Resulting in more efficient add/search
 * operations, however code size is also increased.
 *************************************************/
#define PRTOS_CONFIG_USE_SORTED_LISTS



/*********** Memory Settings ***********/

/*************************************************
 * @macro: PRTOS_CONFIG_MEM_WIDTH_8_BITS
 *
 * @desc: Target memory is 8 bits wide.
 *
 * @unit: bits
 *************************************************/
/*#define PRTOS_CONFIG_MEM_WIDTH_8_BITS */

/*************************************************
 * @macro: PRTOS_CONFIG_MEM_WIDTH_16_BITS
 *
 * @desc: Target memory is 16 bits wide.
 *
 * @unit: bits
 *************************************************/
/*#define PRTOS_CONFIG_MEM_WIDTH_16_BITS */

/*************************************************
 * @macro: PRTOS_CONFIG_MEM_WIDTH_32_BITS
 *
 * @desc: Target memory is 32 bits wide.
 *
 * @unit: bits
 *************************************************/
#define PRTOS_CONFIG_MEM_WIDTH_32_BITS

/*************************************************
 * @macro: PRTOS_CONFIG_OS_HEAP_SIZE_BYTES
 *
 * @desc: Size (in bytes) of the statically allocated
 * OS Heap. The OS Heap is split into a part used by the
```

```
 * kernel, and a part that can be used by the user. The size
 * of the user heap is defined by
 * @ref(PRTOS_CONFIG_USER_HEAP_SIZE_BYTES).
 * Kernel heap size = OS heap size - user heap size.
 *
 * @dtype: U32_t
 * @unit: bytes
 ********************************************************/
#define PRTOS_CONFIG_OS_HEAP_SIZE_BYTES                 4096

/*********************************************************
 * @macro:  PRTOS_CONFIG_USER_HEAP_SIZE_BYTES
 *
 * @desc: Size (in bytes) of the User part
 * of the OS Heap.
 *
 * @dtype: U32_t
 * @unit: bytes
 ********************************************************/
#define PRTOS_CONFIG_USER_HEAP_SIZE_BYTES               128

/*********************************************************
 * @macro: PRTOS_CONFIG_N_USER_POOLS
 *
 * @desc: Number of pools available to the user to allocate
 * blocks of memory. Pools are statically allocated.
 ********************************************************/
#define PRTOS_CONFIG_N_USER_POOLS                       1


/*********************************************************
 * @macro: PRTOS_CONFIG_ENABLE_MEMORY_PROTECTION
 *
 * @desc: Enables protection of each pool by means of
 * padding and a checksum.
 ********************************************************/
 /* !!!NOT YET AVAILABLE IN V 0.4.X!!! */
#define PRTOS_CONFIG_ENABLE_MEMORY_PROTECTION           0


/*********** Module Settings ***********/
/* On PRTOS_CONFIG_USE_<MODULE>_EVENT_* directives:
 * Defined events will be generated by
 * their respective objects and are
 * available for subscription by tasks.
 * The available events per modules can be
 * found in their respective header files
 * in the include folder. */


/* Task Module settings */

/*********************************************************
 * @macro: PRTOS_CONFIG_STANDARD_STACK_SIZE_BYTES
 *
 * @desc: Defines the standard stack size in bytes.
 *
```

```
 * @dtype: U32_t
 * @unit: Bytes
 *****************************************************/
#define PRTOS_CONFIG_STANDARD_STACK_SIZE_BYTES                  0

/*****************************************************
 * @macro: PPRTOS_CONFIG_REAL_TIME_TASK_DEADLINE_DEFAULT_MS
 *
 * @desc: The Real-Time Task Deadline
 * indicates the amount of time a real-time task is allowed
 * to be delayed before starting execution.
 * This value is used a default, the deadline can be changed
 * using TaskDeadlineSet.
 *
 * @dtype: U32_t
 * @unit: Milliseconds
 *****************************************************/
#define PRTOS_CONFIG_REAL_TIME_TASK_DEADLINE_DEFAULT_MS 15

//#define PRTOS_CONFIG_USE_TASK_EVENT_EXECUTE_EXIT
//#define PRTOS_CONFIG_USE_TASK_EVENT_CREATE_DELETE
#define PRTOS_CONFIG_USE_TASK_EVENT_SUSPEND


/* Timer Module Settings. */

/*****************************************************
 * @macro: PRTOS_CONFIG_ENABLE_SOFTWARE_TIMERS
 *
 * @desc: Enable the software Timer module.
 *****************************************************/
#define PRTOS_CONFIG_ENABLE_SOFTWARE_TIMERS                     1

/*****************************************************
 * @macro: PRTOS_CONFIG_TIMER_INTERVAL_RESOLUTION_MS
 *
 * @desc: Defines the number of OS ticks that occur before
 * all software timers are updated. A higher prescaler results
 * in software timers with a lower resolution e.g.
* Prescaler 5 => resolution: 5 * OS tick period. On the
* other hand it also reduces CPU load.
*
* @dtype: U32_t
* @unit: Milliseconds
 *****************************************************/
#define PRTOS_CONFIG_TIMER_INTERVAL_RESOLUTION_MS               50

#define PRTOS_CONFIG_USE_TIMER_EVENT_OVERFLOW
#define PRTOS_CONFIG_USE_TIMER_EVENT_START_STOP_RESET


/* Mailbox Module Settings. */
/*****************************************************
 * @macro: PRTOS_CONFIG_ENABLE_MAILBOXES
 *
 * @desc: Enable the Mailbox module.
 *****************************************************/
```

```
#define PRTOS_CONFIG_ENABLE_MAILBOXES                                   1

#define PRTOS_CONFIG_USE_MAILBOX_EVENT_POST_PEND

/* Semaphore Module Settings. */
/*****************************************************
 * @macro: PRTOS_CONFIG_ENABLE_SEMAPHORES
 *
 * @desc: Enable the Semaphore module.
 *****************************************************/
#define PRTOS_CONFIG_ENABLE_SEMAPHORES                                  0

#define PRTOS_CONFIG_USE_SEM_EVENT_ACQUIRE_RELEASE

/* Eventgroup Module Settings. */
/*****************************************************
 * @macro: PRTOS_CONFIG_ENABLE_EVENTGROUPS
 *
 * @desc: Enable the Eventgroup module.
 *****************************************************/
#define PRTOS_CONFIG_ENABLE_EVENTGROUPS 1

#define PRTOS_CONFIG_USE_EVENTGROUP_EVENT_FLAG_SET
#define PRTOS_CONFIG_USE_EVENTGROUP_EVENT_FLAG_CLEAR


/* Ring-buffer Module Settings. */
/*****************************************************
 * @macro: PRTOS_CONFIG_ENABLE_RINGBUFFERS
 *
 * @desc: Enable the Ringbuffer module.
 *****************************************************/
#define PRTOS_CONFIG_ENABLE_RINGBUFFERS                                 1

#define PRTOS_CONFIG_USE_RINGBUFFER_EVENT_DATA_IN_OUT
#define PRTOS_CONFIG_USE_RINGBUFFER_EVENT_EMPTY_FULL
#define PRTOS_CONFIG_USE_RINGBUFFER_EVENT_PURGE



/*********** Utility Settings ***********/

/*****************************************************
 * @macro: PRTOS_CONFIG_ENABLE_CPULOAD_CALC
 *
 * @desc: Enable CPU load calculation for each task.
 *****************************************************/
#define PRTOS_CONFIG_ENABLE_CPULOAD_CALC                                0

/*****************************************************
 * @macro: PRTOS_CONFIG_USE_TASKNAMES
 *
 * @desc: Use task names. A task can be assigned a
 * human-readable name with TaskGenericNameSet.
```

```
 ******************************************************/
/* #define PRTOS_CONFIG_USE_TASKNAMES */

/*******************************************************
 * @macro: PRTOS_CONFIG_TASK_NAME_LENGTH_CHARS
 *
 * @desc: Defines the maximum length of a task name in chars.
 *
 * @dtype: U8_t
 * @unit: characters
 ******************************************************/
#define PRTOS_CONFIG_TASK_NAME_LENGTH_CHARS                    20

/*******************************************************
 * @macro: PRTOS_CONFIG_USE_CONVERT_LIB_IN_APP
 *
 * @desc: Use the Convert library in the application.
 ******************************************************/
#define PRTOS_CONFIG_USE_CONVERT_LIB_IN_APP

/*******************************************************
 * @macro: PRTOS_CONFIG_ENABLE_WATCHDOG
 *
 * @desc: Enable the Watchdog timer.
 ******************************************************/
#define PRTOS_CONFIG_ENABLE_WATCHDOG                           0


/*********** Logging and Debugging Settings ***********/

/*******************************************************
 * @macro: PRTOS_CONFIG_ENABLE_LOGGING
 *
 * @desc: Enable the Logger module.
 ******************************************************/
#define PRTOS_CONFIG_ENABLE_LOGGING                            1

/*******************************************************
 * @macro: PRTOS_CONFIG_DEBUG_SERIAL_BAUD_RATE_BPS
 *
 * @desc: Defines the baud rate of the Debug serial port in bits per second.
 *
 * @dtype: U32_t
 * @unit: Bits per second (bps)
 ******************************************************/
#define PRTOS_CONFIG_DEBUG_SERIAL_BAUD_RATE_BPS            115200

/*******************************************************
 * @macro: PRTOS_CONFIG_USE_NEWLIB
 *
 * @desc: Use C newlib for standard IO.
 ******************************************************/
#define PRTOS_CONFIG_USE_NEWLIB                                1

/*******************************************************
 * @macro: PRTOS_CONFIG_ENABLE_LOG_EVENT
 *
```

```
 * @desc: Enable Event logging.
 ********************************************************/
#define PRTOS_CONFIG_ENABLE_LOG_EVENT                    0

/*********************************************************
 * @macro: PRTOS_CONFIG_ENABLE_LOG_INFO
 *
 * @desc: Enable Info logging.
 ********************************************************/
#define PRTOS_CONFIG_ENABLE_LOG_INFO                     1

/*********************************************************
 * @macro: PRTOS_CONFIG_ENABLE_LOG_ERROR
 *
 * @desc: Enable Error logging.
 ********************************************************/
#define PRTOS_CONFIG_ENABLE_LOG_ERROR                    1

/*********************************************************
 * @macro: PRTOS_CONFIG_ENABLE_LOG_DEBUG
 *
 * @desc: Enable Debug logging.
 ********************************************************/
#define PRTOS_CONFIG_ENABLE_LOG_DEBUG                    1

/* !!!NOT YET AVAILABLE IN V 0.4.1!!! */
/* #define PRTOS_CONFIG_USE_LOGGER_MODE_FILE */
```

# 5. Types

This chapter describes all the data types defined by and used throughout the OS. All type-definitions are suffixed with _t. The types discussed in this chapter are defined in the following header files: *StdTypes.h*, *OsTypes.h* and *IdType.h*. Some modules might define additional types, module specific types are discussed in their respective module chapter.

## 5.1. Standard types

All standard types are defined in *StdTypes.h*.

*Table 5.1: Standard types*

| Definition | Description |
|---|---|
| *U8_t* | Unsigned 8-bit integer. |
| *U16_t* | Unsigned 16-bit integer. |
| *U32_t* | Unsigned 32-bit integer. |
| *U64_t* | Unsigned 64-bit integer. |
| *S8_t* | Signed 8-bit integer. |
| *S16_t* | Signed 16-bit integer. |
| *S32_t* | Signed 32-bit integer. |
| *S64_t* | Signed 65-bit integer. |

## 5.2. ID type

Defined in *IdType.h*. All objects existing contained the OS lists are assigned an ID. This object ID is passed when making a system call to indirectly reference the intended object.
The ID type is a 32-bits unsigned integer, and consists of the two parts shown in Table 5.2.

*Table 5.2: ID type parts*

| Name | Bit range | Remarks |
|---|---|---|
| ID Group | 31:24 | Indicates the group an object belongs to. All available ID groups are defined by the *IdGroup_t* enumeration. See Table 5.3. |
| Sequence number | 23:0 | Unsigned integer. |

*Table 5.3: IdGroup_t*

| Name |
|---|
| Memory pool |
| Task |
| Timer |
| Eventgroup |
| Semaphore |
| Mailbox |
| Ringbuffer |

## 5.3. OS types

All OS types are defined in *OsTypes.h*. The *OsTypes.h* header includes *StdTypes.h* and *IdType.h*.

*Table 5.4: OS types*

| | |
|---|---|
| *MemBase_t* | RAM Memory base type. Width is defined by ref *CONFIG_MEM_WIDTH_X_BITS*. |
| *Prio_t* | Priority type. Valid range: 1-5. |
| *OsVer_t* | OS Version. Format: 0x0041 = V 0.4.1 |
| *OsResult_t* | System call result. More info see Section 5.3.1. |
| *Task_t* | Task handler entry point. Each task is assigned a handler function of type *Task_t* upon creation. This handler will be called by the kernel. |

### 5.3.1. OS Result

*Table 5.5: OS Result*

| Mnemonic | Value | Description |
|---|---|---|
| OS_RES_CRIT_ERROR | -6 | The system call has caused a critical error. The reason is further specified in the API description of the caller. |
| OS_RES_ERROR | -5 | The system call has caused a error. The reason is further specified in the API description of the caller. |
| OS_RES_RESTRICTED | -3 | The system call violated access rights. |
| OS_RES_FAIL | -3 | The system call failed. The reason is further specified in the API description of the caller. |
| OS_RES_INVALID_ARGUMENT | -2 | Argument(s) passed to the system call is/are invalid. |
| OS_RES_INVALID_ID | -1 | One or more IDs passed to the system call were invalid. This could either be a non existing object or the wrong ID type. |
| OS_RES_OK | 0 | The system call was successful. |
| OS_RES_LOCKED | 1 | The system call attempted to access a resource that is locked. |
| OS_RES_TIMEOUT | 2 | The system call has timed out. |
| OS_RES_POLL | 3 | The system call's caller is now polling for a event. |
| OS_RES_EVENT | 4 | An event has occurred. |

# 6. Port

The Port abstracts the hardware of the platform the OS is running on. When PriorRTOS is ported to a new platform the API defined by the Port must be implemented. The Port layer consists of the following parts:

- Core: Abstracts the OS timer and interrupts. Used by the Core module. File(s): *PortCore.h/c*.
- Watchdog: Abstracts the Watchdog timer. Used by the Core module. File(s): *PortWdt.h/c*.
- Debug: Abstracts the Debug serial port. Used by the Logger module. File(s): *PortDebug.h/c*.

## 6.1. *Port* API

### 6.1.1. PortSuperVisorModeEnable

void PortSuperVisorModeEnable(void)

Descrip.: Enables supervisor mode.

### 6.1.2. PortSuperVisorModeDisable

void PortSuperVisorModeDisable(void)

Descrip.: Disables supervisor mode.

### 6.1.3. PortGlobalIntEnable

void PortGlobalIntEnable(void)

Descrip.: Enables global interrupts.

### 6.1.4. PortGlobalIntDisable

void PortGlobalIntDisable(void)

Descrip.: Disables global interrupts.

### 6.1.5. PortOsTimerInit

void PortOsTimerInit(U16_t prescaler, U16_t ovf)

Descrip.: Initializes the OS Timer with given settings. The OS Timer must generate a periodic interrupt, the OS Tick. Inside the Interrupt Service Routine the OS Tick function (defined below) must be called.

Input: `(U16_t) prescaler`: Timer prescaler value.
`(U16_t) ovf`: Timer overflow value.

### 6.1.6. PortOsTimerEnable

void PortOsTimerEnable(void)

Descrip.: Enables the OS Timer.

### 6.1.7. PortOsTimerDisable

void PortOsTimerDisable(void)

Descrip.: Disables the OS Timer.

### 6.1.8. PortOsTimerTicksGet

U32_t PortOsTimerTicksGet(void)

Descrip.: Returns the current number of ticks of the OS Timer.
Return: `(U32_t)` ticks
`Any`: valid number of ticks.

### 6.1.9. PortOsTimerTicksSet

void PortOsTimerTicksSet(U32_t ticks)

Descrip.: Sets the number of ticks of the OS Timer.
Input: `(U32_t) ticks`: Number of ticks.

### 6.1.10. PortOsTimerTicksReset

void PortOsTimerTicksReset(void)

Descrip.: Sets the number of ticks of the OS Timer to 0.

### 6.1.11. PortOsIntInit

void PortOsIntInit ( IrqPriority_t os_tick_irq_prio )

Descrip.: Initializes the OS Interrupt at the given IRQ priority.
Input: `(IrqPriority_t) os_tick_irq_prio`: OS Tick IRQ priority level.

### 6.1.12. PortOsIntEnable

void PortOsIntEnable(void)

Descrip.: Enables the OS Interrupt.

### 6.1.13. PortOsIntDisable

void PortOsIntDisable(void)

Descrip.: Enables the OS Interrupt.

### 6.1.14. PortOsIntFlagClear

void PortOsIntFlagClear(void)

Descrip.: Clears the OS Interrupt flag.

### 6.1.15. OsTick

extern void OsTick(void)

Descrip.: OS Tick function. Must be called by the OS Tick Interrupt Service Routine.

### 6.1.16. PortDebugUartInit

void PortDebugUartInit(U32_t baud_rate)

Descrip.: Initializes the Debug serial port at the specified baud rate.
Input: `(U32_t) baud_rate`: Debug serial baud rate in bits per second.

### 6.1.17. PortDebugUartWriteString

U32_t PortDebugUartWriteString(char str)

Descrip.: Writes a null-terminated string to the Debug serial port.
Input: `(char ) str`: Null-terminated char array.
Return: `(U32_t)` number of chars written.
`0`: if no chars were written due to an error or busy port.
`Other`: number of chars written.

### 6.1.18. PortDebugUartWriteChar

U8_t PortDebugUartWriteChar(char c)

Descrip.: Writes a single char to the Debug serial port.
Input: `(char) c`: Character.
Return: `(U8_t)` Characters written.
`0`: if no char was written due to an error or busy port.
`1`: character was written.

### 6.1.19. PortDebugCallbackReadChars

void PortDebugCallbackReadChars(char c, U32_t n)

Descrip.: Reads the received characters. Must be called when characters were received on the Debug serial port.
Input: `(char ) chars`: Character array.
`(U32_t) n`: Number of chars received.

# 7. Event System

The Event System is used throughout the OS to communicate the occurrence of events. An event can range from object creation to a list that gets unlocked. All event types can be viewed in Table 7.1. Events follow publish-subscribe semantics, an event is emitted by a source and all registered tasks will receive it.

*Table 7.1: Event types*

| Macro | Event type | Description |
|---|---|---|
| *EVENT_TYPE_ACCESS* | Access | Events related to read/write access of a particular object. |
| *EVENT_TYPE_STATE_CHANGE* | State change | Events related to state changes of a particular object. E.g. Timer transitioned from Running state to Stopped state. |
| *EVENT_TYPE_CREATE* | Create | Published when an object of the specified type was created. |
| *EVENT_TYPE_DELETE* | Delete | Published when a particular object was deleted. |
| *EVENT_TYPE_EXCEPTION* | Exception | Published when an exception is thrown. |

## 7.1. Emitting

The occurrence of an event is communicated by emitting an event. An event can be emitted by any object in the OS. Every module defines its own set of events, for example the Timer module defines *TIMER_EVENT_OVERFLOW*. An event can only be emitted if its *PRTOS_CONFIG_USE_<MODULE>_EVENT_<EVENT* macro is defined. Emitted events always consist of a source ID and the event specifier.

## 7.2. Registering

Tasks can register to events in a blocking (aka waiting) or non-blocking (aka polling) manner. Events can be registered to either manually using **TaskWait** and **TaskPoll**, or implicitly by system calls.

## 7.3. Handling

When an event occurs and is emitted the registered tasks will receive this event. If the task was not active it is activated and scheduled to executed. Once a task is executing, the task handles occurred events.

## 7.4. Characteristics

Events offer a generic and very flexible way of communicating throughout the system. However, there are a few things to be kept in mind when designing an application:

- When a event is emitted, it only exists for a defined amount of OS ticks (*PRTOS_CONFIG_EVENT_LIFE_TIME_TICKS*). After that the event will be destroyed, unless it is reset by another occurrence.
- An event CANNOT be added to the same list twice. Instead the event's lifetime or timeout is reset.
- An Interrupt Service Routine (ISR) CANNOT register to events, only tasks can.
- Events CANNOT be received if a task is in the Disabled state.
- Events CAN still be received when a task is in the scheduling queue or during execution.

# 8. Scheduling policy

Tasks in a system typically have different timing requirements. This has been reflected in the scheduling policy and the way priorities are assigned to tasks.

A task is assigned one of five categories, indicating its timing requirements; OS, real-time, high, medium or low. More information on the task categories can be found in Table 8.1 below. Additionally the task is assigned a priority level ranging from 1 through 5, where 5 is the highest priority within its respective category.

*Table 8.1: Task categories*

| Category | Description |
|---|---|
| OS(5) | Restricted category only to be used by the kernel. |
| Real-Time(4) | Tasks in this category have to make their deadline and are essential to their system. The maximum amount of time allowed between activation and execution is defined by its deadline. Real-Time tasks are scheduled using a policy that consists of Shortest Deadline First (SDF) combined with its minor priority level. |
| High (3) | High priority tasks are still very important to their system, but missing a deadline would not result in critical errors. |
| Medium (2) | Medium priority tasks are less important to the stability of their system and are allowed to miss their deadline. The time between the deadline and the actual execution are minimized by the scheduler. |
| Low (1) | Low priority tasks are the least important to their system. Tasks in this category are allowed to miss their deadline to allow higher priority tasks to execute. |

# 9. OS Control

## 9.1. Description

This paragraph contains the OS control API functions e.g. initialization, scheduler locking. These functions are to be used with care, since they allow (almost) direct control over the OS, wrong usage can result in system crashes.
All API functions in this module contain the prefix *Os*.

## 9.2. *Os* API

### 9.2.1. OsInit

OsResult_t OsInit(OsResult_t result_optional)

Descrip.: Initializes the Prior RTOS kernel. This function has to be called before any other Prior API function.
Output: `(OsResult_t ) result_optional`: Result of optional module initialization.
Return: `(OsResult_t)` sys call result
`OS_RES_OK`: all essential modules were initialized successfully.
`OS_RES_ERROR`: if one of the essential modules was not initiated successfully. It is recommended to call OsReset or hard-reset the target.

### 9.2.2. OsStart

void OsStart (Id_t start_task_id)

Descrip.: Starts the OS tick and scheduler. The first task to be executed can be specified by start_task_id. When an error occurs while loading the specified start task, the Idle task is loaded instead. Note: This function only returns when OsStop is called or when a critical error occurred.
Input: `(Id_t) start_task_id`: ID of the first task to be executed.

### 9.2.3. OsStop

void OsStop(void)

Descrip.: Stops the OS tick and scheduler. The currently executing task will either finish execution (in cooperative mode) or be suspended (in pre-emptive mode). All tasks and other objects will be deleted.

### 9.2.4. OsFrequencyGet

U16_t OsFrequencyGet(void)

Descrip.: Returns the current OS frequency in Hz.
Return: `(U16_t)` Current OS frequency
`0`: if an error occurred.
`Other`: for valid frequencies.

### 9.2.5. OsVersionGet

OsVer_t OsVersionGet(void)

Descrip.: Returns the OS version e.g. 0x0101 = V1.01. OsVer_t may be converted to a string using ConvertOsVersionToString.

Return: (OsVer_t) Current OS version
0: if an error occurred.
Other: for valid versions.

### 9.2.6. OsRunTimeGet

OsResult_t OsRunTimeGet(OsRunTime_t runtime)

Descrip.: Copies the current OS runtime to the runtime array.
Input: (OsRunTime_t) runtime: Array initialized with OS_RUN_TIME_INIT.
Output: (OsRunTime_t) runtime: runtime[0] = hours, runtime[1] = microseconds.
Return: (OsResult_t) sys call result
OS_RES_OK: if the operation was successful.
OS_RES_ERROR: if the array did NOT comply with the requirements stated in the description.

### 9.2.7. OsRunTimeMicrosDelta

U32_t OsRunTimeMicrosDelta(U32_t us)

Descrip.: Returns the difference between us and the current micros.
Input: (U32_t) us: Earlier moment in time in microseconds
Return: (U32_t) Delta micros
0: if the calculation could not be performed.
Other: valid delta value.

### 9.2.8. OsRunTimeMicrosGet

U32_t OsRunTimeMicrosGet(void)

Descrip.: Returns the 'microseconds' component of the OS runtime.

### 9.2.9. OsRunTimeHoursGet

U32_t OsRunTimeHoursGet(void)

Descrip.: Returns the 'hours' component of the OS runtime.
Return: (U32_t) OS runtime hours.
Any: valid hours.

### 9.2.10. OsTickPeriodGet

U32_t OsTickPeriodGet(void)

Descrip.: Returns the OS tick period in microseconds. Microseconds can be converted to milliseconds using ConvertUsToMs.
Return: (U32_t) OS tick period in us
0: if an error occurred.
Other: for valid tick periods.

### 9.2.11. OsTasksTotalGet

U32_t OsTasksTotalGet(void)

Descrip.: Returns the total number of tasks currently present in the system.
Return: (U32_t) Total number of tasks.
0: if an error occurred.
Other: for valid number of tasks.

### 9.2.12. OsTasksActiveGet

U32_t OsTasksActiveGet(void)

Descrip.: Returns the number of active tasks currently present in the system. A task is considered active if it occupies one of the following states: TASK_STATE_RUNNING, TASK_STATE_ACTIVE or TASK_STATE_CRITICAL.

Return: `(U32_t)` Number of active tasks.
`0`: if an error occurred.
`Other`: for valid number of tasks.

### 9.2.13. OsEventsTotalGet

U32_t OsEventsTotalGet(void)

Descrip.: Returns the number of emitted events in the system at that moment.

Return: `(U32_t)` Total number of events.
`0`: if an error occurred.
`Other`: for valid number of events.

### 9.2.14. OsTaskExists

bool OsTaskExists(Id_t task_id)

Descrip.: Validates if the passed ID belongs to an existing task.

Return: `(bool)` Validation result.
`false`: if the ID does not belong to an existing task.
`true`: if the ID does belong to an existing task.

### 9.2.15. OsCurrentTaskGet

Id_t OsCurrentTaskGet(void)

Descrip.: Returns the ID of the current running task.

Return: `(Id_t)` Task ID
`OS_ID_INVALID`: error occurred.
`Other`: valid current task ID.

### 9.2.16. OsCritSectBegin

void OsCritSectBegin(void)

Descrip.: Locks the scheduler and disables interrupts. This function should ONLY be used in critical sections that require precise timing. This function may be called recursively throughout other functions, as long as every OsCritSectBegin call is paired with a OsCritSectEnd call within the same function scope.

### 9.2.17. OsCritSectEnd

void OsCritSectEnd(void)

Descrip.: Unlocks the scheduler and enables interrupts. This function should be called at the end of the critical code section. This function may be called recursively throughout other functions, as long as every OsCritSectBegin call is paired with a OsCritSectEnd call within the same function scope.

### 9.2.18. OsIsrBegin

void OsIsrBegin(void)

Descrip.: Informs the kernel that a user ISR is currently executing. This prevents the kernel from switching tasks during the executing of an interrupt. Note that the OS tick interrupt will keep occurring (if its priority is higher). The kernel will only switch tasks when all interrupts have finished execution. Note that a OsIsrBegin call HAS to be paired with a OsIsrEnd call within the same ISR scope.

### 9.2.19. OsIsrEnd

void OsIsrEnd(void)

Descrip.: Informs the kernel that a user ISR has finished executing and is allowed to switch tasks. The kernel will only switch tasks when all interrupts have finished execution. Note that a OsIsrBegin call HAS to be paired with a OsIsrEnd call within the same ISR scope.

### 9.2.20. OsIsrNestCountGet

S8_t OsIsrNestCountGet(void)

Descrip.: Returns the number of Interrupt Service Routines that are nested and have called OsIsrBegin.

Return: `(S8_t)` Nest count
`-1`: Cannot access the nest counter at this moment.
`0`: No nesting.
`0ther`: Nesting level.

### 9.2.21. OsSchedulerLock

void OsSchedulerLock(void)

Descrip.: Locks the scheduler preventing the kernel from scheduling NEW tasks for execution. Already scheduled tasks will still execute. When the execution queue is empty the Idle task will be executed. This function may be called recursively throughout other functions, as long as every OsSchedulerLock call is paired with a OsSchedulerUnlock call within the same function scope.

### 9.2.22. OsSchedulerUnlock

void OsSchedulerUnlock(void)

Descrip.: Unlocks the scheduler allowing it to scheduler new tasks if the schedule lock counter is equal to zero i.e. all nested locks have executed their respective unlocks. This function may be called recursively throughout other functions, as long as every OsSchedulerLock call is paired with a OsSchedulerUnlock call within the same function scope.

### 9.2.23. OsSchedulerIsLocked

bool OsSchedulerIsLocked(void)

Descrip.: Returns the scheduler lock state.

Return: `(bool)` Scheduler lock state
`false`: Not locked.
`true`: Locked.

# 10. Tasks

## 10.1. Description

Tasks are an essential part of the application running on the OS. Tasks provide a way to split the application code, and provide execution context for other system calls. Each task can be scheduled and executed independent from other tasks (assuming no dependencies exist). A task is created using **TaskCreate**, the following attributes must be specified:

- the task handler function, must adhere to the *Task_t* signature
- a category and priority (see scheduling policy described in Chapter 8)
- a creation parameter, described in Table 10.1
- its stack size, currently unused
- a optional pointer and value argument that are passed to the task handler function when executing

Currently the scheduler does not support saving task context (CPU register and stack), this has a number of consequences:

- Task code must be implemented in a re-entrant way.
- Tasks must yield execution voluntarily.
- All system calls implement a non-blocking event registration.
- **TaskPoll** must be used to manually register to events, as **TaskWait** is currently unavailable.

All API functions related to Tasks are prefixed with *Task*.

### 10.1.1. Creation parameters

*Table 10.1: Task creation parameters*

| Parameter | Definition | Description |
|---|---|---|
| *TASK_PARAMETER_NONE* | 0x00 | This task has no parameters. |
| *TASK_PARAMETER_START* | 0x01 | This task is promoted to the active state after creation. |
| *TASK_PARAMETER_ESSENTIAL* | 0x02 | This task is essential to the system and cannot therefore not be deleted. |

### 10.1.2. Events

*Table 10.2: Task events*

| Event | Event type | Description |
|---|---|---|
| *TASK_EVENT_SUSPEND* | State change | Emitted when the task is suspended. |
| *TASK_EVENT_EXECUTE* | State change | Emitted when the task is executed. |
| *TASK_EVENT_EXIT* | State change | Emitted when the task exits. |
| *TASK_EVENT_DISABLE* | State change | Emitted when the task is disabled. |

## 10.2. *Task* API

### 10.2.1. TaskCreate

Id_t TaskCreate(Task_t handler, TaskCat_t category, Prio_t priority , const void p_arg, U32_t v_arg)

Descrip.: Creates a Task for the given handler. The task is now able to be scheduled based on its category and priority. The arguments are passed to task upon execution.
Input: `(Task_t) handler`: Task handler function.
`(TaskCat_t) category`: Task category TASK_CAT_HIGH, TASK_CAT_REALTIME.
`(Prio_t) priority`: Task priority
`(U8_t) param`: Task creation parameter. E.g. (TASK_PARAM_ESSENTIAL | TASK_PARAM_NO_PREEM)
Use TASK_PARAM_NONE to pass no parameters.
`(U32_t) stack_size`: Task stack size in bytes. Pass TASK_STD_STACK_SIZE to get the configured standard stack size.
`(const void) p_arg`: Pointer task argument. Passed to the task when executed.
`(U32_t) v_arg`: Value task argument. Passed to the task when executed.
Return: `(Id_t)` Task ID
`INVALID_ID`: if an error occurred during task creation.
`Other`: if successful.

### 10.2.2. TaskDelete

OsResult_t TaskDelete(Id_t task_id)

Descrip.: Deletes the Task matching the task_id. The task cannot be scheduled anymore after calling this function. Note: task_id will be set to INVALID_ID to avoid illegal use of the deleted task.
Input: `(Id_t ) task_id`: Task ID. NULL for current task.
Return: `(OsResult_t)` sys call result
`OS_RES_OK`: if operation was successful.
`OS_RES_ERROR`: if the task handler was not found in any of the lists.

### 10.2.3. TaskIdGet

Id_t TaskIdGet(void)

Descrip.: Returns the ID of the calling task.
Return: `(Id_t)` Task ID
`OS_ID_INVALID`: Error occurred.
`Other`: valid task ID.

### 10.2.4. TaskRealTimeDeadlineSet

OsResult_t TaskRealTimeDeadlineSet(Id_t rt_task_id, U32_t t_ms)

Descrip.: Sets the scheduling deadline of the Real-Time task. This overrides the default deadline defined by CONFIG_REAL_TIME_TASK_DEADLINE_MS_DEFAULT
Input: `(Id_t) rt_task_id`: Real-Time Task ID.
`(U32_t) t_ms`: Deadline in ms.
Return: `(OsResult_t)` sys call result
`OS_RES_OK`: if operation was successful.
`OS_RES_ID_INVALID`: if the given task was not a Real-Time task or if the task ID was an invalid ID (INVALID_ID).
`OS_RES_ERROR`: if the task handler was not found in any of the lists.

### 10.2.5. TaskPrioritySet

OsResult_t TaskPrioritySet(Id_t task_id, Prio_t new_priority)

Descrip.: Assigns a new priority level to the task.
Input: `(Id_t) task_id`: Task ID. INVALID_ID = Running task ID.
`(Prio_t) priority`: New task priority
Return: `(OsResult_t)` sys call result
`OS_RES_OK`: if operation was successful.
`OS_RES_OUT_OF_BOUNDS`: if the new priority was not within bounds (1-5).
`OS_RES_ERROR`: if the task handler was not found in any of the lists.

### 10.2.6. TaskPriorityGet

Prio_t TaskPriorityGet(Id_t task_id)

Descrip.: Returns the priority level of the task.
Input: `(Id_t) task_id`: Task ID. INVALID_ID = Running task ID.
Return: `(Prio_t)` task priority
`0`: task could not be found.
`1-5`: valid task priority.

### 10.2.7. TaskStateGet

TaskState_t TaskStateGet(Id_t task_id)

Descrip.: Returns the task's current state.
Input: `(Id_t) task_id`: Task ID. INVALID_ID = Running task ID.
Return: `(TaskState_t)` task state
`Any`: valid state.

### 10.2.8. TaskRunTimeGet

U32_t TaskRunTimeGet(void)

Descrip.: Returns the current task's runtime (since last task switch) in microseconds.
Return: `(U32_t)`task runtime in us.
`0`: if the task could not be found.
`Other`: if valid.

### 10.2.9. TaskResumeWithVarg

OsResult_t TaskResumeWithVarg(Id_t task_id, U32_t v_arg)

Descrip.: Resumes the task and passes a value argument to it. The value argument will be passed to the task upon execution. NOTE 1: This does NOT work when the task is disabled (TASK_STATE_DISABLED). NOTE 2: If task v_arg is unused, pass 0. NOTE 3: A task CANNOT resume itself.
Input: `(Id_t) task_id`: Task ID.
`(U32_t) v_arg`: Task Value argument.
Return: `(OsResult_t)` sys call result
`OS_RES_OK`: if the task was resumed.
`OS_RES_ERROR`: if the task could not be found.

### 10.2.10. TaskResume

OsResult_t TaskResume(Id_t task_id)

Descrip.: Resumes the task. NOTE 1: This does NOT work when the task is disabled (TASK_STATE_DISABLED). NOTE 2: A task CANNOT resume itself
Input: `(Id_t) task_id`: Task ID.
Return: `(OsResult_t)` sys call result
`OS_RES_OK`: if the task was resumed.

OS_RES_ERROR: if the task could not be found.

### 10.2.11. TaskSuspendSelf

TaskSuspendSelf()

Descrip.: Calling task will suspend execution.

### 10.2.12. TaskSleep

OsResult_t TaskSleep(U32_t t_ms)

Descrip.: Calling task will sleep for the specified amount of time after exiting. After the sleep-timer expires, the task is automatically woken and executed.
Input: (U32_t) t_ms: Sleep time in milliseconds.
Return: (OsResult_t) sys call result
OS_RES_OK: if the sleep timer was created.
OS_RES_ERROR: if the task could not be found.

### 10.2.13. TaskPoll

OsResult_t TaskPoll(Id_t object_id, U32_t event, U32_t timeout_ms, bool add_poll)

Descrip.: The task will poll for the event emitted by the specified object in a NON-BLOCKING fashion. When this event occurs the task will be activated. If the event does not occur within the specified time, the event times out and the task will be activated to handle the timeout.
Input: (Id_t) object_id: ID of the event generating object.
(U32_t) event: Event to listen to.
(U32_t) timeout_ms: Event timeout in milliseconds. If OS_TIMEOUT_INFINITE is passed, the task will wait indefinitely.
(bool) add_poll: Add a new event poll is the event has occurred or is not yet polling.
Return: (OsResult_t) sys call result
OS_RES_POLL: if the event is being polled.
OS_RES_EVENT: if the polled event has occurred.
OS_RES_TIMEOUT: if the timeout expired.
OS_RES_FAIL: if the task is not polling the event (and !add_poll).
OS_RES_ERROR: if an error occurred.

### 10.2.14. TaskWait

OsResult_t TaskWait(Id_t object_id, U32_t event, U32_t timeout_ms)

Descrip.: Listens the task to the specified event emitted by the specified object in a BLOCKING fashion. When this event occurs the task will be activated. If the event does not occur within the specified time, the event subscription times out and the task will be activated to handle the timeout. Arguments:
Input: (Id_t) object_id: ID of the event generating object. If INVALID_ID the task will be listened to all.
(U32_t) event: Event to listen to.
(U32_t) timeout_ms: Event timeout in milliseconds. If OS_TIMEOUT_INFNITE is passed, the task will wait indefinitely.
Return: (OsResult_t) sys call result
OS_RES_EVENT: if the event has occurred.
OS_RES_TIMEOUT: if the timeout expired.
OS_RES_ERROR: if an error occurred.

### 10.2.15. TaskJoin

OsResult_t TaskJoin(Id_t task_id, U32_t timeout)

Descrip.: The calling task will wait/poll for the specified task to be deleted. Calling task will only wait/poll until the timeout.

Input: `(Id_t) task_id`: Task ID to join with.
`(U32_t) timeout`: Amount of time to wait/poll for the join event.

Return: `(OsResult_t)` sys call result
`OS_RES_EVENT`: if the event has occurred.
`OS_RES_TIMEOUT`: if the timeout expired.
`OS_RES_ERROR`: if the task could not be found.

### 10.2.16. TASK_INIT_BEGIN

TASK_INIT_BEGIN()

Descrip.: Initializes the variables required by the event handlers. All user code between _BEGIN and _END will only be executed once.

### 10.2.17. TASK_INIT_END

TASK_INIT_END()

Descrip.: Indicates the end of the TASK_INIT block. Must be used in combination with TASK_INIT_BEGIN().

# 11. Memory Management

## 11.1. Description

Dynamic allocation in real-time systems is frowned upon by some programmers because it could introduce possible sources of instability like fragmentation, dangling pointers and memory leaks. Prior's Memory Management module was designed to provide dynamic allocation while keeping the chances of said instabilities arising low.
The OS heap is a statically allocated part of the RAM of fixed size, $S_{osh}$, to avoid collision with the stack and native heap, it also provides a better estimation of the software's memory usage at compile time. The OS heap is made of two parts: the Kernel and Object (K&O) heap and the User heap. The user heap size is defined by $S_{uh}$. The K&O heap size will be equal to $S_{osh}$ - $S_{uh}$. The User heap can be split into a number of pools, $N_p$. Each pool's size is configured upon creation using **MemPoolCreate**. The concept described above is illustrated in Figure 11.1. Allocating memory in a pool has the following advantages:

- It can be protected by padding and a pool-checksum. If an executing task using allocated memory overwrites any of the padding, it will be dispatched and disabled. The padding is repaired afterwards.
- It is guaranteed to be zeroed when initially allocated.
- It is guaranteed to be consecutive.
- It can be managed by pool operations allowing for sorting, de-fragmenting, formatting and moving.

All API functions related to Memory Management are prefixed with *Mem*.
$S_{osh}$ = *PRTOS_CONFIG_OS_HEAP_SIZE_BYTES*
$S_{uh}$ = *PRTOS_CONFIG_USER_HEAP_SIZE_BYTES*
$N_p$ = *PRTOS_CONFIG_N_POOLS*

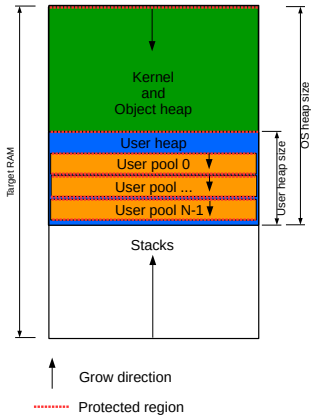


*Figure 11.1: Memory layout*

## 11.2. *Mem* API

### 11.2.1. MemPoolCreate

Id_t MemPoolCreate(U32_t size)

Descrip.: Creates a memory pool on the OS heap if the amount of space specified is available. MemPoolCreate will return an invalid ID (0xFFFF) if the operation failed. Arguments:
Input: `(U32_t) size`: Size to allocate for the pool.
Return: `(Id_t)` Pool ID.
`INVALID_ID`: if the pool was not created.
`Other`: Valid ID if successful

### 11.2.2. MemPoolDelete

void MemPoolDelete(Id_t pool_id)

Descrip.: Formats and deletes the pool. Arguments:
Input: `(Id_t) pool_id`: Pool to delete.
Return: `(OsResult_t)` sys call result
`OS_RES_OK`: if the pool was deleted.
`OS_RES_ID_INVALID`: if the pool ID does not exist.
`OS_RES_RESTRICTED`: if a pool was accessed without the correct privileges.

### 11.2.3. MemPoolFormat

OsResult_t MemPoolFormat(Id_t pool_id)

Descrip.: Formats the pool. All data will be lost.
Input: `(Id_t) pool_id`: Pool to format.
Return: `(OsResult_t)` sys call result
`OS_RES_OK`: if the pool was formatted.
`OS_RES_ID_INVALID`: if the pool ID does not exist.
`OS_RES_RESTRICTED`: if a pool was accessed without the correct privileges.

### 11.2.4. MemPoolDefrag

void MemPoolDefrag(Id_t pool_id)

Descrip.: De-fragments the pool. This will result in a more continuous pool, allowing for larger allocations. NOTE: Pool Allocation table required.
Input: `(Id_t) pool_id`: Pool to de-fragment.

### 11.2.5. MemPoolMove

OsResult_t MemPoolMove (Id_t src_pool, Id_t dst_pool)

Descrip.: Moves the source pool to the destination pool. The source pool will be zeroed. Returns OS_RES_OK if operation was successful, OS_RES_FAIL if the destination pool is too small.
Input: `(Id_t) src_pool`: To-move pool ID
`(Id_t) dst_pool`: Destination pool ID
Return: `(OsResult_t)` sys call status
`OS_RES_OK`: if move was successful.
`OS_RES_RESTRICTED`: if a pool was accessed without the correct privileges.

### 11.2.6. MemPoolFreeSpaceGet

U32_t MemPoolFreeSpaceGet(Id_t pool_id)

Descrip.: Returns the amount of space (in bytes) available in the pool.
Input: `(Id_t) pool_id`: Pool ID.

### 11.2.7. MemOsHeapFreeSpaceGet

U32_t MemOsHeapFreeSpaceGet(void)

Descrip.: Returns the amount of space (in bytes) available on the OS Heap.

### 11.2.8. MemAlloc

void MemAlloc (Id_t pool_id, U32_t size)

Descrip.: Dynamically allocates memory in given pool with specified size. Allocated memory may be freed or reallocated. If allocation fails, MemAllocDynamic will return NULL.
Input: `(Id_t) pool_id`: ID of the pool where the memory will be allocated.
`(U32_t) size`: Size to allocate
Return: `(void )` pointer to memory.
`NULL`: if allocation failed.
`Other`: Valid pointer if successful

### 11.2.9. MemReAlloc

OsResult_t MemReAlloc (Id_t cur_pool_id, Id_t new_pool_id, void ptr, U32_t new_size)

Descrip.: Re-Allocates memory in given pool with specified size. The allocation may be moved by passing a different pool ID. If allocation fails, MemReAlloc returns OS_RES_FAIL. In this case the memory will still remain allocated in the current pool.
Input: `(Id_t) cur_pool_id`: Current pool ID of the to-reallocate memory
`(Id_t) new_pool_id`: New pool ID of the to-reallocate memory. If moving the allocation across memory pools is not desired, either pass the same pool ID for new_pool_id as cur_pool_id or pass INVALID_ID as new_pool_id.
Output: `(void) ptr`: Pointer to existing allocation
`(U32_t) new_size`: Size to allocate
Return: `(OsResult_t)` sys call result
`OS_RES_OK`: if re-allocation was successful.
`OS_RES_FAIL`: if the requested block was too large.
`OS_RES_OUT_OF_BOUNDS`: if the pool ID is not part of the pool memory space.
`OS_RES_RESTRICTED`: if a pool was accessed without the right privileges.

### 11.2.10. MemFree

OsResult_t MemFree (void ptr)

Descrip.: Frees the specified piece of allocated memory,returning it to the pool. Freed memory will be set to 0. The pointer to the freed memory will be implicitly set to NULL.
Input: `(void) ptr`: Pointer to the allocation pointer
Return: `(OsResult_t)` sys call result
`OS_RES_OK`: if freeing was successful.
`OS_RES_NULL_POINTER`: if the memory pointer equals NULL
`OS_RES_OUT_OF_BOUNDS`: if the pool ID is not part of the pool memory space.
`OS_RES_RESTRICTED`: if a pool was accessed without the right privileges.

### 11.2.11. MemAllocSizeGet

U32_t MemAllocSizeGet(void ptr)

Descrip.: Returns the size of the allocated piece of memory.
Input: `(void ) ptr`: Pointer to allocated memory.

Return: `(U32_t)` allocation size.
`0`: if the operation failed.
`Other`: Valid allocation size.

# 12. Timers

## 12.1. Description

Timer objects are software timers with a PRTOS_CONFIG_TIMER_RESOLUTION_MS resolution. Every timer is updated by the TimerUpdate kernel task. When a timer overflows it will emit an event if PRTOS_CONFIG_USE_TIMER_EVENT_OVERFLOW is defined. Add refs to config.
All functions related to Timers are prefixed with *Timer*.

### 12.1.1. Timer parameter

Every timer has a parameter value that is used to configure the operation mode.
The timer parameters are defined by the TIMER_PARAMETER_* macros.

- TIMER_PARAMETER_ON (bit 0): ON, timer will be ON after creation if this bit is 1.
- TIMER_PARAMETER_PERIODIC (bit 1): Periodic, timer will not be deleted after an overflow if bit is 1.
- TIMER_PARAMETER_AR (bit 2): Auto-Reset, timer is reset upon overflow if this bit is 1. If this is not the case, the timer will stay in the waiting state until reset manually.
- TIMER_PARAMETER_ITR_SET(itr) (bits 3-7): Contain the number of timer iterations. This number decreases after every overflow, the timer will be deleted if this number equals zero and the Periodic-bit is 0.

### 12.1.2. States

A timer can occupy on of three states; *TIMER_STATE_STOPPED*, *TIMER_STATE_RUNNING* and *TIMER_STATE_WAITING*. Figure X shows the state diagram of these states and their respective transitions.
*TIMER_STATE_STOPPED*: The timer is in an inactive state, this is the default state after creation.
*TIMER_STATE_RUNNING*: The timer is in an active state where its counter is incremented automatically.
*TIMER_STATE_RUNNING*: The timer is waiting to be reset.

### 12.1.3. Events

*Table 12.1: Timer events*

| Event | Event type | Description |
|---|---|---|
| *TIMER_EVENT_OVERFLOW* | State change | Published when the Timer's counter has reached its set interval value. |
| *TIMER_EVENT_START* | State change | Published when the Timer was started. |
| *TIMER_EVENT_STOP* | State change | Published when a Timer was stopped. |
| *TIMER_EVENT_RESET* | State change | Published when a Timer was reset. |

## 12.2. *Timer* API

### 12.2.1. TimerCreate

Id_t TimerCreate(U32_t interval, U8_t parameter, TimerOverflowCallback_t overflow_callback, void context)

Descrip.: Creates a timer that will overflow after the specified interval. If the TIMER_PARAMETER_ON flag is set, the timer will start after creation. The number of iterations before auto-delete can be specified through the use of the parameter. Upon overflowing the timer will do the following: - If PRTOS_CONFIG_USE_TIMER_EVENT_OVERFLOW: Emit a timer overflow event. - If overflow_callback != NULL: Call the overflow callback. - If TIMER_PARAMETER_ON = 1: Restart the timer. - If TIMER_PARAMETER_AR = 1: Automatically reset. - If TIMER_PARAMETER_PERIODIC = 0: Decrement the number of iterations left. - If iterations left = 0: Delete the timer.

Input: `(U32_t) interval_us`: Timer interval in microseconds(us), 0xFFFFFFFF is illegal.
`(U8_t) parameter`: Timer parameter.
`(TimerOverflowCallback_t) overflow_callback`: Called when the timer overflows.
`(void ) context`: Opaque context pointer, argument of the overflow_callback. Set NULL if unused.

Return: `(Id_t)` Timer ID
`INVALID_ID`: if creation failed.
`Other`: if the timer was created.

### 12.2.2. TimerDelete

OsResult_t TimerDelete(Id_t timer_id)

Descrip.: Deletes the specified timer and sets timer_id to INVALID_ID if the operation is successful.

Output: `(Id_t ) timer_id`: ID of the timer to delete. Will be set to INVALID_ID.

Return: `(OsResult_t)` sys call result
`OS_RES_OK`: if the timer was successfully deleted.
`OS_RES_ERROR`: if the timer could not be found.

### 12.2.3. TimerStop

OsResult_t TimerStop(Id_t timer_id)

Descrip.: Stops the specified timer, it will not start unless TimerStart is called. The timer is transitioned to the stopped-state. The current ticks on the timer are REMOVED, to save the ticks use TimerPause.

Input: `(Id_t) timer_id`: ID of the timer to stop.

Return: `(OsResult_t)` sys call result
`OS_RES_OK`: if the timer was successfully stopped.
`OS_RES_ERROR`: if the timer could not be found.

### 12.2.4. TimerStart

OsResult_t TimerStart(Id_t timer_id)

Descrip.: Starts the specified timer.

Input: `(Id_t) timer_id`: ID of the timer to start.

Return: `(OsResult_t)` sys call result
`OS_RES_OK`: if the timer was successfully started.
`OS_RES_ERROR`: if the timer could not be found.

### 12.2.5. TimerPause

OsResult_t TimerPause(Id_t timer_id)

Descrip.: Pauses the specified timer, it will not start unless TimerStart is called. The timer is transitioned to the stopped-state. The current ticks on the timer are SAVED, to remove ticks use TimerStop or TimerReset. Arguments:
Input: `(Id_t) timer_id`: ID of the timer to pause.
Return: `(OsResult_t)` sys call result
`OS_RES_OK`: if the timer was successfully paused.
`OS_RES_ERROR`: if the timer could not be found.

### 12.2.6. TimerReset

OsResult_t TimerReset(Id_t timer_id)

Descrip.: Resets the timer's current ticks.
Input: `(Id_t) timer_id`: ID of the timer to reset.
Return: `(OsResult_t)` sys call result
`OS_RES_OK`: if the timer was successfully reset.
`OS_RES_ERROR`: if the timer could not be found.

### 12.2.7. TimerStartAll

void TimerStartAll(void)

Descrip.: Starts all timers that exist at the time of the call.

### 12.2.8. TimerStopAll

void TimerStopAll(void)

Descrip.: Stops all timers that exist at the time of the call.

### 12.2.9. TimerResetAll

void TimerResetAll(void)

Descrip.: Reset all timers that exist at the time of the call.

### 12.2.10. TimerStateGet

TmrState_t TimerStateGet (Id_t timer_id)

Descrip.: Returns the current state of the timer.
Input: `(Id_t) timer_id`: Timer ID.
Return: `(TmrState_t)` current state
`TIMER_STATE_STOPPED`:
`TIMER_STATE_WAITING`:
`TIMER_STATE_RUNNING`:
`TIMER_STATE_INVALID`: if the timer could not be found.

### 12.2.11. TimerTicksGet

U32_t TimerTicksGet (Id_t timer_id)

Descrip.: Returns the current value of the timer's counter.
Input: `(Id_t) timer_id`: Timer ID.
Return: `(U32_t)` counter value
`TIMER_INTERVAL_ILLEGAL`: if the timer could not be found.
`Other`: valid counter value.

### 12.2.12. TimerIntervalSet

void TimerIntervalSet(Id_t timer_id, U32_t new_interval_us)

Descrip.: Sets a new interval for the timer. The timer is NOT implicitly reset.
Input: `(Id_t) timer_id`: Timer ID.
`(U32_t) new_interval_us`: New Timer interval in microseconds(us).

### 12.2.13. TimerIntervalGet

U32_t TimerIntervalGet (Id_t timer_id)

Descrip.: Returns the current interval of the timer in us.
Input: `(Id_t) timer_id`: Timer ID.
Return: `(U32_t)` timer interval
`TIMER_INTERVAL_ILLEGAL`: if the timer could not be found.
`Other`: valid interval.

### 12.2.14. TimerIterationsGet

U8_t TimerIterationsGet(Id_t timer_id)

Descrip.: Returns the current amount of iterations left on the timer.
Input: `(Id_t) timer_id`: Timer ID.
Return: `(U8_t)` timer iterations
`0`: if the timer could not be found.
`1-31`: for a valid number of iterations.

### 12.2.15. TimerIterationsSet

OsResult_t TimerIterationsSet(Id_t timer_id, U8_t iterations)

Descrip.: Sets the amount of iterations left on the timer. The value should be between 1 and 31.
Input: `(Id_t) timer_id`: Timer ID.
`(U8_t) iterations`: Number of iterations, 1-31.
Return: `(OsResult_t)` Sys call result
`OS_RES_OK`: if the operation was successful.
`OS_RES_ERROR`: if the timer could not be found.
`OS_RES_OUT_OF_BOUNDS`: if the iteration value was > 31 or 0.

### 12.2.16. TimerParameterGet

U8_t TimerParameterGet (Id_t timer_id)

Descrip.: Returns the current parameter of the timer.
Input: `(Id_t) timer_id`: Timer ID.
Return: `(U8_t)` sys call result
`0xFF`: if the timer could not be found.
`Other`: valid parameter.

### 12.2.17. TimerParameterSet

U8_t TimerParameterSet (Id_t timer_id)

Descrip.: Sets a new parameter for the timer.
Input: `(Id_t) timer_id`: Timer ID.
`(U8_t) parameter`: New timer parameter. 0xFF is illegal.

# 13. Semaphores

## 13.1. Description

Semaphores are small objects that can be used to lock resources that lack a locking system, such a shared (raw) memory or a peripheral. These locks can be used to keep tasks from accessing a shared resource simultaneously. This keeps the shared data in between users in sync. Prior currently distinguishes only 1 kind of semaphore, the mutex. This is a classic lock, in the sense that it can only be locked once. By calling **SemaphoreAcquire** the lock is acquired, the lock is unlocked, or released using **SemaphoreRelease**.
All API functions related to Semaphores are prefixed with *Semaphore*.

### 13.1.1. Events

*Table 13.1: Semaphore events*

| Event | Event type | Description |
|---|---|---|
| *SEMAPHORE_EVENT_ACQUIRE* | State change | Emitted when the semaphore is acquired. |
| *SEMAPHORE_EVENT_RELEASE* | State change | Emitted when the semaphore is released. |

## 13.2. *Semaphore* API

### 13.2.1. SemaphoreCreate

Id_t SemaphoreCreate(U8_t sem_type, U8_t max_count)

Descrip.: Creates a semaphore of the specified type. If this type is not SEM_TYPE_MUTEX_BINARY the maximum count will be set to max_count.
Input: `(U8_t) sem_type`: Semaphore type. Currently only SEM_TYPE_MUTEX_BINARY.
`(U8_t) max_count`: Maximum allowed recursive acquires.
Return: `(Id_t)` Semaphore ID.
`INVALID_ID`: if the creation failed.
`Other`: valid ID if the semaphore was created.

### 13.2.2. SemaphoreDelete

OsResult_t SemaphoreDelete(Id_t sem_id)

Descrip.: Deletes the semaphore and sets sem_id to INVALID_ID.
Input: `(Id_t ) sem_id`: Semaphore ID.
Return: `(OsResult_t)` sys call result
`OS_RES_OK`: if the semaphore was deleted.
`OS_RES_LOCKED`: if the Semaphore is still acquired by one or multiple users.
`OS_RES_ERROR`: if the semaphore was not found.

### 13.2.3. SemaphoreAcquire

OsResult_t SemaphoreAcquire(Id_t sem_id, U32_t timeout)

Descrip.: Attempts to acquire the semaphore.
Input: `(Id_t ) sem_id`: Semaphore ID.
`(U32_t) timeout`: Timeout in ms.
Return: `(OsResult_t)` sys call result
`OS_RES_OK`: if the semaphore was acquired.
`OS_RES_LOCKED`: if the Semaphore is still acquired by one or multiple users.
`OS_RES_ERROR`: if the semaphore was not found.

### 13.2.4. SemaphoreRelease

OsResult_t SemaphoreRelease(Id_t sem_id)

Descrip.: Releases the acquired semaphore.
Input: `(Id_t ) sem_id`: Semaphore ID.
Return: `(OsResult_t)` sys call result
`OS_RES_OK`: if the semaphore was released.
`OS_RES_LOCKED`: if the semaphore was not acquired.
`OS_RES_ERROR`: if the semaphore was not found.

### 13.2.5. SemaphoreCountSet

OsResult_t SemaphoreCountSet(Id_t sem_id, U8_t count)

Descrip.: Sets a new maximum acquire count for the semaphore.
Input: `(Id_t) sem_id`: Semaphore ID.
`(U8_t) count`: New max. acquire count.
Return: `(OsResult_t)` sys call result
`OS_RES_OK`: if the new count was set.
`OS_RES_ERROR`: if the semaphore was not found.

### 13.2.6. SemaphoreCountGet

OsResult_t SemaphoreCountGet(Id_t sem_id)

Descrip.: Returns the current acquire count of the semaphore. Arguments:
Input: `(Id_t) sem_id`: Semaphore ID.
Return: `(U8_t)` acquire count
`Any`: valid count.

### 13.2.7. SemaphoreCountReset

OsResult_t SemaphoreCountReset(Id_t sem_id)

Descrip.: Forces the semaphore to be released by all users, setting the acquire count to 0. Tasks that have acquired the semaphore at that moment will be suspended.
Input: `(Id_t) sem_id`: Semaphore ID.
Return: `(OsResult_t)` sys call result
`OS_RES_OK`: if the count was reset.
`OS_RES_ERROR`: if the semaphore was not found.

# 14. Mailboxes

## 14.1. Description

A mailbox is an array that is owned by one of more tasks, of which each index is individually addressable. The data at each of the addresses has a pend counter, which enables read confirmation by each of the owner tasks. Data cannot be written to an address if the pend counter is not 0. Writing to a mailbox can be done by any task using **MailboxPost**. However, only the owner(s) may read the data using **MailboxPend**. Every mailbox has a width defined by the *MailboxBase_t* type and length size, the size specified upon creation.
All API functions related to Mailboxes are prefixed with *Mailbox*.

### 14.1.1. Events

*Table 14.1: Mailbox events*

| Event | Event type | Description |
|---|---|---|
| *MAILBOX_EVENT_POST(addr)* | Access | Emitted when data is posted at the specified address. |
| *MAILBOX_EVENT_PEND(addr)* | Access | Emitted when data is pended from the specified address. |
| *MAILBOX_EVENT_POST_ALL* | Access | Emitted when data is posted at any address. |

## 14.2. *Mailbox* API

### 14.2.1. MailboxCreate

Id_t MailboxCreate(U8_t mailbox_size, Task_t owners[], U8_t n_owners)

Descrip.: Creates a mailbox of the specified size with given owners. Only the owners are allowed pend from the created mailbox. The pend counter of each address is set to the number of owners upon posting. Note the width of each mailbox address is defined by MailboxBase_t (U8_t by default).
Input: `(U8_t) mailbox_size`: Number of addresses reserved for this mailbox.
`(Id_t) owner_ids`: Array of owner task IDs.
`(U8_t) n_owners`: Number of owners in the owners list.
Return: `(Id_t)` Mailbox ID
`INVALID_ID`: if the creation failed.
`Other`: valid ID if the mailbox was created.

### 14.2.2. MailboxDelete

OsResult_t MailboxDelete(Id_t mailbox_id)

Descrip.: Deletes the specified mailbox. mailbox_id is set to INVALID_ID.
Output: `(Id_t ) mailbox_id`: ID of the mailbox to delete.
Return: `(OsResult_t)` sys call result
`OS_RES_OK`: if the mailbox was deleted.
`OS_RES_ERROR`: if the mailbox could not be found.

### 14.2.3. MailboxPost

OsResult_t MailboxPost(Id_t mailbox_id, U8_t address, MailboxBase_t data, U32_t timeout)

Descrip.: Post data in the mailbox starting at the address. Data cannot be overwritten when it has not been pended by its owner i.e. the pend counter is not 0. Any task can post in any mailbox.
Input: `(Id_t) mailbox_id`: ID of the mailbox to post in.

(U8_t) address: Address where data will be posted.
(MailboxBase_t) data: Data to be posted.

Return: (OsResult_t) sys call result
OS_RES_OK: if the mailbox was deleted.
OS_RES_ERROR: if the mailbox could not be found.
OS_RES_LOCKED: if the pend counter of one of the addresses within the specified range is not 0.
OS_RES_OUT_OF_BOUNDS: if the address is not part of the mailbox address range.

### 14.2.4. MailboxPend

OsResult_t MailboxPend(Id_t mailbox_id, U8_t base_address, MailboxBase_t data, U8_t len)

Descrip.: Pend data with given length from the mailbox starting at the base-address. Pending data decrements the pend counter. Only owner tasks may pend from the mailbox.

Input: (Id_t) mailbox_id: ID of the mailbox to pend from.
(U8_t) base_address: Starting address where data will be pended.
(MailboxBase_t ) data: Pointer to the array where the data will be copied to.

Return: (OsResult_t) sys call result
OS_RES_OK: if the mailbox was deleted.
OS_RES_ERROR: if the mailbox could not be found.
OS_RES_LOCKED: if the pend counter is already 0.
OS_RES_RESTRICTED: if the pending task is not an owner.
OS_RES_OUT_OF_BOUNDS: if the address is not part of the mailbox address range.

### 14.2.5. MailboxPendCounterGet

OsResult_t MailboxPendCounterGet(Id_t mailbox_id, U8_t address)

Descrip.: Returns the pend counter of the given address. A mailbox- address can only be posted to if the pend counter is 0. A mailbox- address can only be pended from if the pend counter is >0.

Input: (Id_t) mailbox_id: ID of the mailbox.
(U8_t) address: Mailbox address.

# 15. Eventgroups

## 15.1. Description

An Eventgroup is a set of 8 flags that can be individually set, cleared and checked. Each flag can publish events on set and clear operations. Eventgroups are versatile in usage due to their simplicity. All API functions related to Eventgroups are prefixed with *Eventgroup*.

### 15.1.1. Events

*Table 15.1: Eventgroup events*

| Event | Event type | Description |
|---|---|---|
| *EVENTGROUP_EVENT_FLAG_SET(flag_mask)* | State change | Published when the specified flag is set. |
| *EVENTGROUP_EVENT_FLAG_CLEAR(flag_mask)* | State change | Published when the specified flag is cleared. |

## 15.2. *Eventgroup* API

### 15.2.1. EventgroupCreate

Id_t EventgroupCreate(void)

Descrip.: Creates a new eventgroup. Each eventgroup register contains 8 flags that can individually set or cleared.
Return: `(Id_t)` Eventgroup ID
`INVALID_ID`: if an error occurred during creation.
`Other`: if the eventgroup was successfully created.

### 15.2.2. EventgroupDelete

Id_t EventgroupDelete(Id_t eventgroup_id)

Descrip.: Deletes an existing eventgroup. The evengroup_id will be set to INVALID_ID if the operation is successful.
Input: `(Id_t ) eventgroup_id`: Eventgroup ID.
Return: `(OsResult_t)` sys call result
`OS_RES_OK`: if the eventgroup was successfully deleted.
`OS_RES_ERROR`: if the eventgroup could not be found.

### 15.2.3. EventgroupFlagsSet

void EventgroupFlagsSet(Id_t eventgroup_id, U8_t mask)

Descrip.: Sets the masked flags in the eventgroup register. E.g. if mask = (EVENTGROUP_FLAG_MASK_3 | EVENTGROUP_FLAG_MASK_7), bit 3 and 7 will be set.
Input: `(Id_t) eventgroup_id`: Eventgroup ID.
`(U8_t) mask`: Event flag mask.
Return: `(OsResult_t)` sys call result
`OS_RES_OK`: if the eventgroup was successfully deleted.
`OS_RES_ERROR`: if the eventgroup could not be found.

### 15.2.4. EventgroupFlagsClear

OsResult_t EventgroupFlagsClear(Id_t evengroup_id, U8_t mask)

Descrip.: Clears the masked flags in the eventgroup register. E.g. if mask = (EVENTGROUP_FLAG_MASK_0 | EVENTGROUP_FLAG_MASK_4), bit 0 and 4 will be cleared.

Input: `(Id_t) eventgroup_id`: Eventgroup ID.
`(U8_t) mask`: Event flag mask.
Return: `(OsResult_t)` sys call result
`OS_RES_OK`: if the eventgroup was successfully deleted.
`OS_RES_ERROR`: if the eventgroup could not be found.

### 15.2.5. EventgroupFlagsGet

U8_t EventgroupFlagsGet(Id_t evengroup_id, U8_t mask)

Descrip.: Returns the eventgroup flags specified in the mask.
Input: `(Id_t) eventgroup_id`: Eventgroup ID.
`(U8_t) mask`: Event flag mask.

# 16. Ringbuffers

## 16.1. Description

Ring-buffers are part of the Inter-Task Communication system and are ideal for streaming data from one task (or ISR) to another. Ring-buffers lend their name to the fact that the buffer's end is attached to the head, effectively creating a circle. A ring-buffer follows FIFO semantics and manages a read and write pointer that move in the clock-wise direction. The write pointer can only move ot the index before the read pointer and vice versa, ultimately protecting the unread data from being overwritten by new data.

The width of the ring-buffers is defined by *RingbufBase_t*. The size a ring-buffer is configured when creating it. The storage for a ring-buffer can be allocated dynamically upon creation, or an external (static) array can be provided as storage instead. A depiction of a ring-buffer is shown in Figure 16.1. All API functions related to Ring-buffers are prefixed with *Ringbuf*.

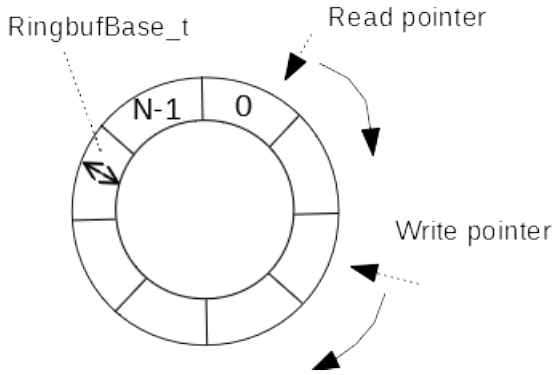


*Figure 16.1: Ring-buffer*

### 16.1.1. Events

*Table 16.1: Ring-buffer events*

| Event | Event type | Description |
|---|---|---|
| *EVENTGROUP_EVENT_DATA_IN* | Access | Emitted when data is written to the ring-buffer. |
| *EVENTGROUP_EVENT_DATA_OUT* | Access | Emitted when data read from the ring-buffer. |
| *EVENTGROUP_EVENT_EMPTY* | State change | Emitted when the ring-buffer was emptied. |
| *EVENTGROUP_EVENT_FULL* | State change | Emitted when the ring-buffer is full. |
| *EVENTGROUP_EVENT_PURGE* | State change | Emitted when the ring-buffer was purged. |

## 16.2. *Ringbuf* API

### 16.2.1. RingbufCreate

Id_t RingbufCreate(RingbufBase_t buffer, U32_t size)

Descrip.: Creates a ring-buffer of given size. Note that the width of each element in the buffer is defined by RingbufBase_t (U8_t by default). An external buffer can provided to store the data. If this buffer is not provided it will be allocated internally (dynamic allocation is used).
Input: `(RingbufBase_t ) buffer`: External buffer. NULL for internal allocation.
`(U32_t) size`: Ring-buffer size.
Return: `(Id_t)` Ring-buffer ID
`INVALID_ID`: if an error occurred during creation.
`Other`: valid ID if the ring-buffer was created.

### 16.2.2. RingbufDelete

OsResult_t RingbufDelete (Id_t ringbuf_id)

Descrip.: Deletes the specified ring-buffer. ringbuf_id is set to INVALID_ID if the operation is successful.
Input: `(Id_t ) ringbuf_id`: ID of the ring-buffer to delete.
Return: `(OsResult_t)` sys call result
`OS_RES_OK`: if the ring-buffer was deleted.
`OS_RES_ERROR`: if the ring-buffer could not be found.

### 16.2.3. RingbufWrite

OsResult_t RingbufWrite(Id_t ringbuf_id, RingbufBase_t data, U32_t length, U32_t timeout)

Descrip.: Writes data of given length to the ring-buffer. The amount actually written is returned.
Input: `(Id_t) ringbuf_id`: Ring-buffer ID.
`(RingbufBase_t) data`: Array of data.
`(U32_t ) length`: Length of the data.
`(U32_t) timeout`: Timeout is ms.
Output: `(U32_t ) length`: Actual amount written.
Return: `(OsResult_t)` sys call result
`OS_RES_OK`: if data was written.
`OS_RES_FAIL`: if the buffer is empty.
`OS_RES_LOCKED`: if the ringbuffer is already locked for writing.
`OS_RES_ERROR`: if the ringbuffer could not be found.

### 16.2.4. RingbufRead

OsResult_t RingbufRead(Id_t ringbuf_id, RingbufBase_t data, U32_t amount, U32_t timeout)

Descrip.: Reads a given number of data nodes from the ring-buffer. The amount actually read is returned. Read data is copied to the provided target array. The target array has to comply with the pre-conditions stated below.
Input: `(Id_t) ringbuf_id`: Ring-buffer ID.
`(RingbufBase_t) target`: Target data array.Pre-condition target[amount-1] = 0xEF
`(U32_t ) amount`: Amount of data to read.
`(U32_t) timeout`: Timeout is ms.
Output: `(U32_t ) amount`: Actual amount of data read.
Return: `(OsResult_t)` sys call result
`OS_RES_OK`: if data was read.
`OS_RES_LOCKED`: if the ringbuffer is already locked for reading.
`OS_RES_FAIL`: if the buffer is empty.

`OS_RES_OUT_OF_BOUNDS`: if the target array was not compliant.
`OS_RES_ERROR`: if the ringbuffer could not be found.

### 16.2.5. RingbufDump

U32_t RingbufDump(Id_t ringbuf_id, RingbufBase_t target)

Descrip.: Dumps all data present in the ring-buffer in the target array. The amount actually read is returned. The target array has to comply with the pre-conditions stated below.
Input: `(Id_t) ringbuf_id`: Ring-buffer ID.
`(RingbufBase_t) target`: Target data array. Pre-condition target[ring-buffer size-1] = 0xEF
Return: `(U32_t)` Amount read.
`0`: if pre-conditions were not met.
`Other`: valid amount.

### 16.2.6. RingbufFlush

OsResult_t RingbufFlush(Id_t ringbuf_id)

Descrip.: Resets the ring-buffer's read and write locations as well as its current data count resulting in all its initial space becoming available.
Input: `(Id_t) ringbuf_id`: Ring-buffer ID.
Return: `(OsResult_t)` sys call result
`OS_RES_OK`: if the ring-buffer was flushed.
`OS_RES_LOCKED`: if the ring-buffer is locked for reading or writing.
`OS_RES_ERROR`: if the ring-buffer could not be found.

### 16.2.7. RingbufSearch

U32_t RingbufSearch(Id_t ringbuf_id, RingbufBase_t query, U32_t query_length)

Descrip.: Searched the ring-buffer for the given query of given length. The number of occurrences of the query in the buffer is returned.
Input: `(Id_t) ringbuf_id`: Ring-buffer ID.
`(RingbufBase_t) query`: Search query.
`(U32_t) query_length`: Length of the search query.
Return: `(U32_t)` Number of occurrences.
`Any`: valid amount.

### 16.2.8. RingbufDataCountGet

U32_t RingbufDataCountGet(Id_t ringbuf_id)

Descrip.: Returns the amount of data is present in the ring- buffer.
Input: `(Id_t) ringbuf_id`: Ring-buffer ID.
Return: `(U32_t)` Number of data nodes.
`Any`: valid amount.

### 16.2.9. RingbufDataSpaceGet

U32_t RingbufDataSpaceGet(Id_t ringbuf_id)

Descrip.: Returns the amount of space left in the ring-buffer. Arguments:
Input: `(Id_t) ringbuf_id`: Ring-buffer ID.
Return: `(U32_t)` Data space left.
`Any`: valid amount.

# 17. Logger

## 17.1. Description

Enabled by setting at least one of the *PRTOS_CONFIG_ENABLE_LOGGER* settings to 1.
The Logger module enables the programmer to log messages with different purposes at different levels. There are three logging categories, which are indicated by a tag in the log message; info, debug and error. Each of the logging tags will be discussed below.
The logging API is implemented in the shape of logging macros. The advantage of this approach is that all calls to the logging API will be replaced by an empty line if the respective logging category or level has been disabled. This saves ROM or RAM space regardless of compiler optimization flags.
Message format: [TIME-STAMP HH:SS:MS][TAG][Optional 0][Optional n]: Message

- Info: Reserved for the OS. All general information such as the output from the initialization sequence and object creations are logged in this category. Info messages contain a time-stamp and tag along with the message.
- Debug: Can be used most effectively when debugging. A debug message is printed along with a time-stamp, tag, calling function name and line number.
- Error: Is used to log/report errors. An error message is printed along with time-stamp, tag, source file name and line number.

## 17.2. *Log* API

### 17.2.1. LOG_ERROR_NEWLINE

LOG_ERROR_NEWLINE(message, ...)

Descrip.: Logs an error message on a new line. A new line includes the timestamp, tag, function name and line number.

### 17.2.2. LOG_ERROR_APPEND

LOG_ERROR_APPEND(message, ...)

Descrip.: Appends the error message to the current logging line. This appended message will NOT contain a timestamp, tag, function name and line number.

### 17.2.3. LOG_DEBUG_NEWLINE

LOG_DEBUG_NEWLINE(message, ...)

Descrip.: Logs a debug message on a new line. A new line includes the timestamp, tag, source and line number.

### 17.2.4. LOG_DEBUG_APPEND

LOG_DEBUG_APPEND(message, ...)

Descrip.: Appends the debug message to the current logging line. This appended message will NOT contain a timestamp, tag, source and line number.

# 18. Shell

## 18.1. Description

The Shell module is Priors Command Line Interface (CLI) that can be enabled by setting ref *PRTOS_CONFIG_ENABLE_SHELL* to 1.
The Shell is accessible through any desired communication port, the default is UART. Commands are entered as text and suffixed with a backslash N to indicate the end of the command. A typical reply is prefixed with psh> (which can in turn be prefixed by a time-stamp if enabled).

- Spaces are allowed but have no effect on the meaning of the command.
- Command arguments have to be prefixed with -.
- Assigning values to command arguments have to be assigned using =.

Example: *help -c=cfg*, Displays the help menu of the *cfg* command.

In addition to using the built-in Shell commands, it is also possible to add custom commands to the command-set using the *Shell* API.

### 18.1.1. Commands

#### 18.1.1.1. *help*

Displays a list of all available commands if no arguments are passed.
Displays the help menu for the passed command name if this is given.

*Table 18.1: help command argument(s)*

| Optional | Argument name | Format | Command argument | Description |
|---|---|---|---|---|
| X | Command name | String | -c=<string> | Name of the command for which the help menu should be displayed. |

#### 18.1.1.2. *kstat*

Prints the kernel statistics such as current frequency, memory usage, uptime, number of tasks, number of events, execution queue, etc.
The *kstat* command has no arguments.

#### 18.1.1.3. *tstat*

Prints all existing task's statistics such as average run-time and state.
The *tstat* command has no arguments.

#### 18.1.1.4. *run*

The specified task is scheduled to run.

*Table 18.2: run command argument(s)*

| Optional | Argument name | Format | Command argument | Description |
|---|---|---|---|---|
| | Task ID | Hex(4) | -t=<100A>or<100a> | Task to run. |
| X | Force run | Boolean | -f | Forces the scheduler to run this task immediately. |

#### 18.1.1.5. *lsprint*

Prints the details of a list.

*Table 18.3: lsprint command argument(s)*

| Optional | Argument name | Format | Command argument | Description |
|---|---|---|---|---|
| | List | Enum | -l=<list> 0=exeq, 1=tcb, 2=tcbw, | Task to run. |
| X | Force run | Boolean | -f | Forces the scheduler to run this task immediately. |

## 18.2. *Shell* API

### 18.2.1. ShellCommandRegister

OsResult_t ShellCommandRegister(struct ShellCommand command)

Descrip.: Registers a Shell command making available for calling using the CLI.
Input: `(struct ShellCommand ) command`: Initialized ShellCommand structure that defines the command, callbacks and token counts.
Return: `(OsResult_t)` sys call result
`OS_RES_OK`: if the command was registered.
`OS_RES_FAIL`: if the maximum amount of registered commands has been reached.

### 18.2.2. ShellReplyInvalidArgs

void ShellReplyInvalidArgs(char command)

Descrip.: Should be called by the command callbacks when the contents of the arguments are invalid. The user will be informed with this message: "Command '<command>' has invalid arguments".
Input: `(char command) command`: Command in string form.

### 18.2.3. ShellPut

U16_t ShellPut(char message, ...)

Descrip.: Prints a message starting on a new line prefixed with 'psh>'.
Input: `(char ) message`: Message in string form.
`(...) variable arguments`: -
Return: `(U16_t)` Number of characters
`0`: if no characters were printed because the buffer could not process the requested amount.
`Other`: valid number of characters.

### 18.2.4. ShellPutRaw

U16_t ShellPutRaw(char message, ...)

Descrip.: Prints the exact message.
Input: `(char ) message`: Message in string form.
`(...) variable arguments`: -
Return: `(U16_t)` Number of characters
`0`: if no characters were printed because the buffer could not process the requested amount.
`Other`: valid number of characters.

### 18.2.5. ShellPutRawNewline

U16_t ShellPutRawNewline(char message, ...)

Descrip.: Prints the exact message on a new line.
Input: `(char ) message`: Message in string form.
`(...) variable arguments`: -

Return: `(U16_t)` Number of characters
`0`: if no characters were printed because the buffer could not process the requested amount.
`Other`: valid number of characters.

### 18.2.6. ShellCommandFromName

struct ShellCommand ShellCommandFromName(char *name)

Descrip.: Get the ShellCommand struct from the (NULL terminated) command name.
Input: `(char ) name`: NULL terminated command name.
Return: `(struct ShellCommand)` ShellCommand struct belonging to the name.
`NULL`: no ShellCommand matches the name.
`Other`: valid ShellCommand.

# 19. Conversions

## 19.1. Description

The Convert API contains all kinds of conversions that can come in handy, such as the conversion from the OS result type to a string.
All API functions related to conversion are prefixed with *Convert*.

## 19.2. *Convert* API

### 19.2.1. ConvertUsToMs

U32_t ConvertUsToMs(U32_t us)

Descrip.: Convert microseconds (us) to milliseconds (ms).
Input: `(U32_t) us`: Value in microseconds.
Return: `(U32_t)` Value in milliseconds.
`0`: Operation failed.
`Other`: Valid value.

### 19.2.2. ConvertMsToUs

U32_t ConvertMsToUs(U32_t ms)

Descrip.: Convert milliseconds (ms) to microseconds (us).
Input: `(U32_t) ms`: Value in milliseconds.
Return: `(U32_t)` Value in microseconds.
`0`: Operation failed.
`Other`: Valid value.

### 19.2.3. ConvertResultToString

U8_t ConvertResultToString(OsResult_t result, char out_result_str)

Descrip.: Converts a result of type OsResult_t to a null-terminated string.
Input: `(OsResult_t) result`: Result to convert.
Output: `(char ) out_result_str`: Array containing the result string. The array should be at least of size CONVERT_BUFFER_SIZE_RESULT_TO_STRING.
Return: `(U8_t)` number of characters (excluding NULL).
`0`: Operation failed.
`Other`: Valid number of characters.

### 19.2.4. ConvertIntToString

U8_t ConvertIntToString(U32_t integer, char out_int_str )

Descrip.: Converts a integer value to a null-terminated string.
Input: `(U32_t) integer`: Integer value.
Output: `(char ) out_int_str`: Array containing the integer string. out_int_str[N-1] should be initialized at 0x20 (ASCII space) to prevent the conversion from exceeding its bounds.
Return: `(U8_t)` number of characters (excluding NULL).
`0`: Operation failed.
`Other`: Valid number of characters.

### 19.2.5. ConvertIntToBytes

U8_t ConvertIntToBytes(U32_t integer, U8_t num_bytes, U8_t out_bytes)

Descrip.: Convert an integer value into separate bytes. This allows for easy splitting of for instance 32-bits value into 4 bytes. The bytes will be stored with the LSB at 0.
Input: `(U32_t) integer`: Integer value.
`(U8_t) num_bytes`: Number of bytes present in the integer value. 1-4.
Output: `(U8_t ) out_bytes`: Array containing the bytes, should be at least of size num_bytes. out_bytes[0] = LSB. out_bytes[num_bytes-1] = MSB.
Return: `(U8_t)` result.
`0`: Number of bytes has an illegal value i.e. 0 or >4.
`1`: Operation successful.

### 19.2.6. ConvertOsVersionToString

U8_t ConvertOsVersionToString(OsVer_t os_version, char out_os_version_str)

Descrip.: Converts OS version to a null-terminated string of length 7 in the following format: V<M>.<m>.<svn> Major, minor, subversion.
Input: `(OsVer_t) os_version`: OS Version to convert.
Output: `(char ) out_os_version_str`: Array containing the OS version string. The array should be at least of size CONVERT_BUFFER_SIZE_OS_VERSION_TO_STRING.
Return: `(U8_t)` number of characters (excluding NULL).
`0`: Operation failed.
`Other`: Valid number of characters.

### 19.2.7. ConvertHexStringToId

Id_t ConvertHexStringToId(char hex_id_str)

Descrip.: Converts a hexadecimal base string to an ID.
Input: `(char ) hex_id_str`: Hexadecimal base string. E.g 0100142A
Return: `(Id_t)` ID.
`ID_INVALID`: Invalid ID string.
`Other`: Valid ID.

# 20. Guides