Prior RTOS User Manual

Author(s): Dorus van de Spijker

Document history

Rel.	Date	Changes
R01	2017-03-09	First draft

Distribution list

Name	R01
Dorus van de Spijker	x

Open issues

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

Contents

List of Figures

List of Tables

List of Listings

1. References

Mandatory

[1.1]	Title:	SPD Globus Radio Module Hardware
	Author(s):	Free Claessens
	ID, Version, Date:	SPD6983160001, R06, 2017-01-18
	File:	SPD6983160001R06
[1.2]	Title:	SPD Globus Radio Module Software
	Author(s):	Merijn van Eijk
	ID, Version, Date:	SPD6983160016, R06, 2017-02-27
	File:	SPD6983160016R06

Referenced

[2.1]	Title:	WL18x7MOD Wi-Fi and Bluetooth Controller datasheet
	Author(s):	Texas Instruments
	URL:	Datasheet
[2.2]	Title:	CC2564 Bluetooth Controller datasheet
	Author(s):	Texas Instruments
	URL:	Datasheet
[2.3]	Title:	BlueKitchen BTstack supported devices
	Author(s):	BlueKitchen GmBH
	URL:	Supported chipsets

2. Abbreviations and definitions

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

2.1. Abbreviations

API	Application Programming Interface
CLI	Command Line Interface
ID	Identity
OS	Operating System
RTOS	Real-Time Operating System
UART	Universal Asynchronous Receiver Transmitter

2.2. 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).
Type	References to types, macros, fields that are used throughout the OS.
SomeFunction	Function is a part of the Prior RTOS API.

3. Introduction

3.1. 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 AtmelâĂŹs 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.

3.2. Core values

Adaptability

Security and Isolation

3.3. 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 ringbuffers.
- Global Events allowing tasks to Wait and Poll for all possible events.
- Advanced Memory Management giving the programmer protected memory pools allowing dynamic allocations on the statically allocated OS heap.
- 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, peripherals and utilities to tailor the OS to your application.

4. Types

This chapter describes all the data types defined by and used throughout the OS. All type-definitions are suffixed with _t.

4.1. List of types

Table 4.1: Available types

Definition	Description
U8_t	Unsigned 8-bit integer.
U16_t	Unsigned 16-bit integer.
U32_t	Unsigned 32-bit integer.
S8_t	Signed 8-bit integer.
S16_t	Signed 16-bit integer.
S32_t	Signed 32-bit integer.
MemBase_t	RAM Memory base type. Width is defined by ref *CONFIG_MEM_WIDTH_X_BITS*.
Id_t	Object identifier. Used to reference any OS object, more information can be found in Table 4.2.
IdType_t	ID type. Indicates what type of object the ID is belongs to.
Prio_t	Priority type. Valid range: 0-24.
OsVer_t	OS Version. Format: 0x0041 = V 0.4.1
OsResult_t	System call result. More info see ref.
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.

4.2. The Object ID

All objects existing contained the OS lists are assigned an identifier or ID for short. This object ID is passed when making a system call to indirectly reference the meant object.
The ID type is a 16-bits unsigned integer, in most cases, with the exception of Event IDs, consisting of the two parts shown in Table 4.2.

Table 4.2: ID type parts

Name	Bit range	Remarks
Object Type	15:12	Indicates the object type which the ID is assigned to. See Table 4.3.
Unique ID	11:0	Unique identifier within this object type.

Table 4.3: Object type

Name	Value
Memory pool	0x0
Task	0x1
Timer	0x2
Eventgroup	0x3
Semaphore	0x4
Mailbox	0x5
Ringbuffer	0x6

4.3. The OS result type

5. Events

5.1. Publishing

Events are used throughout the OS to communicate the creation and deletion of objects, object access, state changes, exceptions. All event types can be viewed in Table 5.1.
Typically events are generated or Published by OS objects such as Timers (ref).

Table 5.1: 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.

5.2. Subscribing

Tasks can wait for or poll (with possible timeout) these events, this is known as Subscribing. Events can be subscribed to either manually using **TaskWait** and **TaskPoll**, or automatically by other system calls. A subscription requires the publisher's ID (also called the event source), the object specific event e.g. a Timer Overflow event for Timers, Event Flags to enable/disable subscription options these flags are shown in Table 5.2 and an optional timeout time to avoid blocking tasks for longer than necessary.

Table 5.2: Event flags

Macro	**Event flag**	**Description**
EVENT_FLAG_NONE	None	Passed when no event flags should be set.
EVENT_FLAG_ADDRESSED	Addressed	The event will be addressed to the specified task. This means that the task does not need a subscription to these events in order to receive them. However, addressed events cannot be handled using TASK_EVENT_HANDLER sections.
EVENT_FLAG_NO_HANDLER	No handler	Set when the subscribing task does not implement an explicit **TASK_EVENT_HANDLER** section for this event.
EVENT_FLAG_NO_TRIGGER	No trigger	Set to keep the subscribing task from being scheduled when the event was received.
EVENT_FLAG_PERMANENT	Permanent	Indicates that the subscription to this event is permanent i.e. the event subscription will not be removed after receiving it.

5.3. Handling

When an event occurs and is published the subscribed tasks will receive this event and will typically, depending on the subscription flags, be scheduled to execute. Once a triggered task is executing, the task handles occurred events in so called Task Event Handler sections. These sections are defined by a **TASK_EVENT_HANDLER_BEGIN** and **TASK_EVENT_HANDLER_END** pair. Event timeouts can be handled in **TASK_EVENT_HANDLE_TIMEOUT** sections.

5.4. 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 published, 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.

- Any event, published or subscribed, cannot be added to the same list twice. Instead the event's lifetime is reset.
- It IS possible to subscribe to new events within *EVENT_HANDLER* sections.
- It IS NOT possible to subscribe to the same event BEFORE its handler section. This would clear both the occurred event and the newly subscribed one.
- Events CANNOT be received if a task has been transitioned to the Idle state or Disabled state manually, even if it still has valid subscriptions.
- Events CAN still be received when a task is already scheduled and waiting to be executed.

Exampe of event handling

6. Scheduling policy

Tasks in a system typically have different timing requirements. This has been reflected on the scheduling policies and the way of assingning priorities to tasks. Four categories were implemented that indicate the major priority level of each task, indicating its timing requirements; realtime, high, medium, low. More information on the task categories can be found in Table 6.1. Each priority category has 5 minor priority levels ranging from 1 through 5, where 5 is the highest priority within its respective category.

Table 6.1: 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 in 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.

7. Kernel Control

7.1. Description

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

7.2. *Os* API

7.2.1. OsInit

OsInit (OsResult_t result_optional)

Descrip.:	Initializes the Prior RTOS kernel. This function has to be called any other Prior API function.
Output:	`(OsResult_t *) result_optional`: Result of optional module initialization.
Return:	`(OsResult_t)` Sys call result `OS_OK`: all essential modules were initialized successfully. `OS_ERROR`: if one of the essential modules was not initiated successfully.

7.2.2. OsStart

OsStart (Id_t start_task_id)

Descrip.:	Starts the OS tick and scheduler. The first task to be executed be specified by start_task_id. When an error occurs while loading the start task, the Idle task is loaded instead.
Input:	`(Id_t) start_task_id`: ID of the first task to be executed. itialization.

7.2.3. OsFrequencySet

OsFrequencySet(U16_t os_frequency_hz)

Descrip.:	Sets a new OS tick frequency in Hz.
Input:	`(U16_t) os_frequency`: New OS frequency.
Return:	`(OsResult_t)` Sys call result `OS_OK`: if the new frequency was set. `OS_ERROR`: if no configuration could be matched to the requested frequency.

7.2.4. OsFrequencyGet

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.

7.2.5. OsVersionGet

OsVersionGet(void)

Descrip.:	Returns the OS version e.g. 0x0101 = V1.01. OsVer_t may be to a string format using ConvertOsVersionToString.
Return:	`(OsVer_t)` Current OS version `0`: if an error occurred. `Other`: for valid versions.

7.2.6. OsRuntimeGet

OsRuntimeGet(U32_t target)

Descrip.:	Copies the current OS runtime to the target array. Target has to have least 2 elements initialized at 0x00000000;
Output:	`(U32_t *) target`: Target array to return the runtime. target[0] = hours, target[1] = microseconds.

Return: `(OsResult_t)` Sys call result
`OS_OK`: if the operation was successful.
`OS_ERROR`: if the array did NOT comply with the requirements stated in the description.

7.2.7. OsTickPeriodGet

OsTickPeriodGet(void)

Descrip.: Returns the OS tick period in 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.

7.2.8. OsTasksTotalGet

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.

7.2.9. OsTasksActiveGet

OsTasksActiveGet(void)

Descrip.: Returns the number of active tasks currently present in the A task is considered active if it occupies one of the following 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.

7.2.10. OsEventsTotalGet

OsEventsTotalGet(void)

Descrip.: Returns the number of published 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.

7.2.11. OsTaskExists

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.

7.2.12. OsCritSectEnter

OsCritSectEnter(void)

Descrip.: Locks the scheduler and disables interrupts. This function ONLY be used in critical sections like CRC generation/validation sections that require precise timing. function may be called recursively throughout other functions, as long every OsCritSectEnter call is paired with a OsCritSectExit call within same function scope.

7.2.13. OsCritSectExit

OsCritSectExit(void)

Descrip.: Unlocks the scheduler and enables interrupts. This function should called after exiting the critical code section. function may be called recursively throughout other functions, as long every OsCritSectEnter call is paired with a OsCritSectExit call within same function scope.

7.2.14. OsIsrEnter

OsIsrEnter(void)

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

7.2.15. OsIsrExit

OsIsrExit(void)

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

7.2.16. OsSchedulerLock

OsSchedulerLock(void)

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

7.2.17. OsSchedulerUnlock

OsSchedulerUnlock(void)

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

8. Tasks

8.1. *Task* API

8.1.1. TaskCreate

TaskCreate(Task_t handler, TaskCat_t category, Prio_t priority , void args)

Descrip.: Creates a Task Control Block for the given handler. The is now able to be scheduled based on its category and priority. arguments are passed to task upon execution.
Input: `(Task_t) handler`: Task handler function.
`(TaskCat_t) category`: Task category: TASK_CAT_LOW, TASK_CAT_MEDIUM, TASK_CAT_HIGH, TASK_CAT_REALTIME.
`(Prio_t) priority`: Task priority: 1-5 (5 is highest).
`(void*) args`: Task arguments. Passed to the task when executing.
Return: `(Id_t)` Task ID
`INVALID_ID`: if an error occurred during task creation.
`Other`: if successful.

8.1.2. TaskDelete

TaskDelete(Id_t task_id)

Descrip.: Deletes an existing Task Control Block for the given handler. task cannot be scheduled anymore.
Input: `(Id_t *) task_id`: Task ID. INVALID_ID = Running task ID.
Return: `(OsResult_t)` sys call result
`OS_OK`: if operation was successful.
`OS_ERROR`: if the task handler was not found in any of the lists.

8.1.3. TaskDeadlineSet

TaskDeadlineSet(Id_t task_id, U32_t t_ms)

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

8.1.4. TaskPrioritySet

TaskPrioritySet(Id_t task_id, Prio_t new_priority)

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

8.1.5. TaskPriorityGet

TaskPriorityGet(Id_t task_id)

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

8.1.6. TaskCategorySet

TaskCategorySet(Id_t task_id, TaskCat_t new_cat)

Descrip.: Assigns a new category to the given task.
Input: `(Id_t) task_id`: Task ID. INVALID_ID = Running task ID.
`(TaskCat_t) new_cat`: New Task category: TASK_CAT_LOW, TASK_CAT_MEDIUM, TASK_CAT_HIGH, TASK_CAT_REALTIME.
Return: `(OsResult_t)` sys call result
`OS_OK`: if operation was successful.
`OS_OUT_OF_BOUNDS`: if new_cat is not a member of TaskCat_t.
`OS_ERROR`: if the task was not found in any of the lists.
`OS_RESTRICTED`: if the OS category was assigned in user-mode.

8.1.7. TaskCategoryGet

TaskCategoryGet(Id_t task_id)

Descrip.: Returns the given task's category.
Input: `(Id_t) task_id`: Task ID. INVALID_ID = Running task ID.
Return: `(TskCat_t)` task category
`TASK_CAT_OS`: (reserved by OS)
`TASK_CAT_REALTIME`:
`TASK_CAT_HIGH`:
`TASK_CAT_MEDIUM`:
`TASK_CAT_LOW`:

8.1.8. TaskStateGet

TaskStateGet(Id_t task_id)

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

8.1.9. TaskRuntimeGet

TaskRuntimeGet(Id_t task_id)

Descrip.: Returns the given task's average runtime in microseconds.
Input: `(Id_t) task_id`: Task ID. INVALID_ID = Running task ID.
Return: `(U32_t)` task runtime in us.
`0`: if the task could not be found.
`Other`: if valid.

8.1.10. TaskIdGet

TaskIdGet(void)

Descrip.: Returns the ID of the current task.
Return: `(Id_t)` Task ID
`INVALID_ID`: error occurred.
`Other`: valid Task ID.

8.1.11. TaskNotify

TaskNotify(Id_t task_id, void args)

Descrip.: Notifies the task to be come active, it available for scheduling. specified arguments will be passed to the task upon execution.
Input: `(Id_t) task_id`: Task ID.
`(void *) args`: Task arguments.
Return: `(OsResult_t)` sys call result
`OS_OK`: if the task was activated.
`OS_ERROR`: if the task could not be found.

8.1.12. TaskSleep

TaskSleep(U32_t t_ms)

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

8.1.13. TaskPoll

TaskPoll(Id_t object_id, U32_t event, U8_t flags, U32_t timeout_ms, Id_t out_event_id)

Descrip.: Subscribes the task to the specified event published by the specified object in a NON-BLOCKING fashion.
When this event occurs the task will receive it and, depending on the event flags passed, 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.
Input: `(Id_t) object_id`: ID of the event generating object. If INVALID_ID the task will be subscribed to all.
`(U32_t) event`: Event to subscribe to.
`(U8_t) flags`: Flags to be set, use EVENT_FLAG_ macros.
`(U32_t) timeout_ms`: Event timeout in milliseconds. If 0 is passed, the task will wait indefinitely.
Return: `(OsResult_t)` sys call result
`OS_OK`: if the subscription was successful.
`OS_ERROR`: if an error occurred.

9. Memory Management

9.1. Description

Cross-refs

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. PriorâĂŹs 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 *CONFIG_OS_HEAP_SIZE_BYTES* to avoid collision with the stack and native heap, it also provides a better estimation of the softwareâĂŹs memory usage at compile time. The heap can be split into N pools, where N is defined by *CONFIG_N_USER_POOLS*. Each poolâĂŹs size is configured upon creation. The concept described above is depicted in Figure 9.1.
Furthermore, allocating memory in a pool using **MmAlloc** 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 *Mm*.

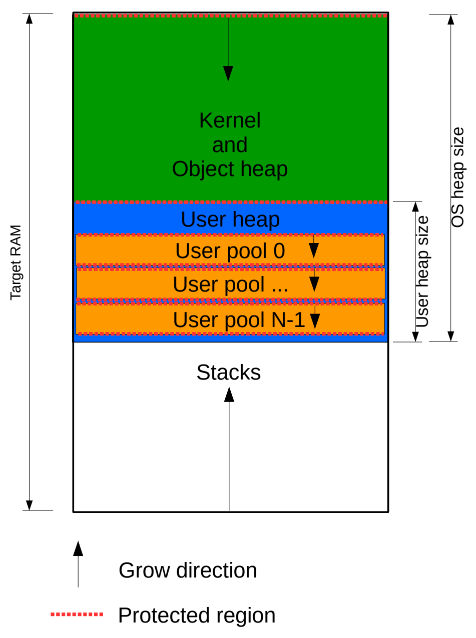

Figure 9.1: Memory layout

9.2. *Mm* API

9.2.1. MmPoolCreate

MmPoolCreate(U32_t size)

Descrip.:	Creates a memory pool on the OS heap if the amount of space is available. MmPoolCreate will return an invalid ID (0xFFFF) if operation failed.
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

9.2.2. MmPoolDelete

MmPoolDelete(Id_t pool_id)

Descrip.:	Formats and deletes the pool.
Input:	`(Id_t) pool_id`: Pool to delete.

9.2.3. MmPoolFormat

MmPoolFormat(Id_t pool_id)

Descrip.: Formats the pool. All data will be lost.
Input: `(Id_t) pool_id`: Pool to format.

9.2.4. MmPoolFreeSpaceGet

MmPoolFreeSpaceGet(Id_t pool_id)

Descrip.: Returns the amount of space (in bytes) available in the pool.
Input: `(Id_t) pool_id`: Pool ID.
Return: `(U32_t)` Free pool space.

9.2.5. MmOsHeapFreeSpaceGet

MmOsHeapFreeSpaceGet(void)

Descrip.: Returns the amount of space (in bytes) available on the Heap.
Return: `(U32_t)` Free OS heap space.

9.2.6. MmAlloc

MmAlloc (Id_t pool_id, U32_t size)

Descrip.: Dynamically allocates memory in given pool with specified size. memory may be freed or reallocated. allocation fails, MmAllocDynamic 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

9.2.7. MmReAlloc

MmReAlloc (Id_t pool_id, void ptr , U32_t size)

Descrip.: Re-Allocates memory in given with specified size. The allocation may be by passing a different pool ID. If allocation MmReAlloc returns OS_FAIL. In this case the memory will 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.
`(U32_t) new_size`: Size to allocate
Return: `(OsResult_t)` sys call result
`OS_OK`: if re-allocation was successful.
`OS_FAIL`: if the requested block was too large.
`OS_OUT_OF_BOUNDS`: if the pool ID is not part of the pool memory space.

9.2.8. MmFree

MmFree (Id_t pool_id, void ptr)

Descrip.: Frees the specified piece of allocated memory,returning it to pool. Freed memory will be set to 0. The pointer to the freed memory be implicitly set to NULL.
Input: `(Id_t) pool_id`: Pool ID of the allocated memory
Return: `(OsResult_t)` sys call result
`OS_OK`: if freeing was successful.
`OS_NULL_POINTER`: if the memory pointer equals NULL
`OS_OUT_OF_BOUNDS`: if the pool ID is not part of the pool memory space.

9.2.9. MmAllocSizeGet

MmAllocSizeGet(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.

10. Timers

10.1. Description

Timer objects are software timers managed by the kernel. Each timer is updated during the OS tick, when a timer overflows it can publish an event. The maximum resolution of each timer depends on the OS tick frequency and the Timer module prescaler. Add refs to config.
All timer API functions have the prefix *Timer*.

10.1.1. Timer parameter

Every timer object contains a timer parameter register that is used to configure the timerâĂŹs operation mode. These registers contain the following settings:
bit 0: ON-bit, timer will be ON after creation if this bit is 1.
bit 1: Periodic-bit, timer will not be deleted after triggering if bit is 1.
bit 2: Auto-Reset-bit, timer will auto-reset if this bit is 1. If this is not the case, the timer will stay in the waiting state until reset manually.
bit 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.

10.1.2. 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.

10.1.3. Events

Table 10.1: 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.

10.2. *Timer* API

10.2.1. TimerCreate

TimerCreate(U32_t interval, U8_t parameter)

Descrip.: Creates a timer that will overflow after the specified If the TIMER_PARAMETER_ON flag is set, the timer will start creation. The number of iterations before auto-delete can be through the use of the parameter. overflowing the timer will do the following: Always: Generate an overflow event. If TIMER_PARAMETER_ON = 1: Start the timer. If TIMER_PARAMETER_AR = 1: Automatically reset. If TIMER_PARAMETER_P = 0: Decrement the number of iterations left. If iterations left = 0: Delete the timer. parameter macros (auto-reset) (on) TIMER_PARAMETER_P(periodic) (sets iterations for parameter) (gets iterations from parameter)

Input: `(U32_t) interval_us`: Timer interval in microseconds(us), 0xFFFFFFFF is illegal.
`(U8_t) parameter`: Timer parameter.

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

10.2.2. TimerDelete

TimerDelete(Id_t timer_id)

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

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

10.2.3. TimerStop

TimerStop(Id_t timer_id)

Descrip.: Stops the specified timer, it will not start unless TimerStart called. The timer is transitioned to the stopped-state. 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_OK`: if the timer was successfully stopped.
`OS_ERROR`: if the timer could not be found.

10.2.4. TimerStart

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_OK`: if the timer was successfully started.
`OS_ERROR`: if the timer could not be found.

10.2.5. TimerPause

TimerPause(Id_t timer_id)

Descrip.: Pauses the specified timer, it will not start unless TimerStart called. The timer is transitioned to the stopped-state. current ticks on the timer are SAVED, to remove ticks use TimerStop or TimerReset.

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

Return: `(OsResult_t)` sys call result
`OS_OK`: if the timer was successfully paused.
`OS_ERROR`: if the timer could not be found.

10.2.6. TimerReset

TimerReset(Id_t timer_id)

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

10.2.7. TimerStartAll

TimerStartAll(void)

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

10.2.8. TimerStopAll

TimerStopAll(void)

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

10.2.9. TimerResetAll

TimerResetAll(void)

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

10.2.10.

TimerGetState (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.

10.2.11.

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.

10.2.12. TimerIntervalSet

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),

10.2.13.

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.

10.2.14. TimerIterationsGet

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.
`131`: for a valid number of iterations.

10.2.15. TimerIterationsSet

TimerIterationsSet(Id_t timer_id, U8_t iterations)

Descrip.: Sets the amount of iterations left on the timer. The value 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_OK`: if the operation was successful.
`OS_ERROR`: if the timer could not be found.
`OS_OUT_OF_BOUNDS`: if the iteration value was > 31 or 0.

10.2.16.

TimerParameterGet (Id_t timer_id)

Descrip.: Returns the current parameter of the timer. parameter bits | 7-3: iterations | 2: auto-reset | 1: periodic | 0: ON | Arguments:
Input: `(Id_t) timer_id`: Timer ID.
Return: `(U8_t)` Sys call result:
`0xFF`: if the timer could not be found.
`Other`: valid parameter.

10.2.17.

TimerParameterSet (Id_t timer_id)

Descrip.: Sets a new parameter for the timer. parameter bits | 7-3: iterations | 2: auto-reset | 1: periodic | 0: ON | Arguments:
Input: `(Id_t) timer_id`: Timer ID.
`(U8_t) parameter`: New timer parameter. 0xFF is illegal.

11. Semaphores

11.1. *Semaphore* API

11.1.1. SemaphoreCreate

SemaphoreCreate(U8_t sem_type, void resource, U8_t max_count)

Descrip.: Creates a semaphore of the specified type. If this type is not the maximum count will be set to max_count. A resource be attached to the semaphore.
Input: `(U8_t) sem_type`: Semaphore type. Currently only SEM_TYPE_MUTEX_BINARY.
`(void *) resource`: Resource to protect.
`(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.

11.1.2. SemaphoreDelete

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_OK`: if the semaphore was deleted.
`OS_LOCKED`: if the Semaphore is still acquired by one or multiple users.
`OS_ERROR`: if the semaphore was not found.

11.1.3. SemaphoreAcquire

SemaphoreAcquire(Id_t sem_id)

Descrip.: Attempts to acquire the semaphore.
Input: `(Id_t ) sem_id`: Semaphore ID.
Return: `(OsResult_t)` sys call result
`OS_OK`: if the semaphore was acquired.
`OS_LOCKED`: if the Semaphore is still acquired by one or multiple users.
`OS_ERROR`: if the semaphore was not found.

11.1.4. SemaphoreAcquire

SemaphoreAcquire(Id_t sem_id)

Descrip.: Attempts to acquire the semaphore. If successful it return the resource.
Input: `(Id_t ) sem_id`: Semaphore ID.
Return: `(void )` resource
`NULL`: the semaphore is still acquired by one or multiple users or the semaphore could not be found.
`!NULL`: pointer to the resource coupled to this semaphore.

11.1.5. SemaphoreRelease

SemaphoreRelease(Id_t sem_id)

Descrip.: Releases the acquired semaphore.
Input: `(Id_t ) sem_id`: Semaphore ID.

Return: `(OsResult_t)` sys call result
`OS_OK`: if the semaphore was released.
`OS_LOCKED`: if the semaphore was not acquired.
`OS_ERROR`: if the semaphore was not found.

11.1.6. SemaphoreCountSet

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_OK`: if the new count was set.
`OS_ERROR`: if the semaphore was not found.

11.1.7. SemaphoreCountGet

SemaphoreCountGet(Id_t sem_id)

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

11.1.8. SemaphoreCountReset

SemaphoreCountReset(Id_t sem_id)

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

12. Mailboxes

12.1. *Mailbox* API

12.1.1. MailboxCreate

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

Descrip.:	Creates a mailbox of the specified size with given Only the owners are allowed pend from the created mailbox. pend counter of each address is set to the number of owners upon posting. the width of each mailbox address is defined by MailboxBase_t (U8_t 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.

12.1.2. MailboxDelete

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_OK`: if the mailbox was deleted.
	`OS_ERROR`: if the mailbox could not be found.

12.1.3. MailboxPost

MailboxPost(Id_t mailbox_id, U8_t base_address, data, U8_t len)

Descrip.:	Post data with given length in the mailbox starting at the Data cannot be overwritten when it has not been pended by its i.e. the pend counter is not 0. task can post in any mailbox.
Input:	`(Id_t) mailbox_id`: ID of the mailbox to post in.
	`(U8_t) base_address`: Starting address where data will be posted.
	`(MailboxBase_t *) data`: Pointer to the array of data to be posted.
	`(U8_t) len`: Length of the data array.
Return:	`(OsResult_t)` sys call result
	`OS_OK`: if the mailbox was deleted.
	`OS_ERROR`: if the mailbox could not be found.
	`OS_LOCKED`: if the pend counter of one of the addresses within the specified range is not 0.
	`OS_OUT_OF_BOUNDS`: if the given data array is too large for the mailbox.

12.1.4. MailboxPend

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

Descrip.:	Pend data with given length from the mailbox starting at the Pending data decrements the pend counter. Only owner 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.
	`(U8_t) len`: Amount of data to be pended.

Return: `(OsResult_t)` sys call result
`OS_OK`: if the mailbox was deleted.
`OS_ERROR`: if the mailbox could not be found.
`OS_LOCKED`: if the pend counter is already 0.
`OS_RESTRICTED`: if the pending task is not an owner.
`OS_OUT_OF_BOUNDS`: if the requested amount of data is larger than the mailbox storage.

12.1.5. MailboxPendCounterGet

MailboxPendCounterGet(Id_t mailbox_id, U8_t address)

Descrip.:	Returns the pend counter of the given address. A mailbox- can only be posted to if the pend counter is 0. A mailbox- 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.
Return:	`(U8_t)` pend counter.

13. Eventgroups

13.1. 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.

13.2. Events

Table 13.1: 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.

13.3. *Eventgroup* API

13.3.1. EventgroupCreate

EventgroupCreate(void)

Descrip.:	Creates a new eventgroup. Each eventgroup register contains 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.

13.3.2. EventgroupDelete

EventgroupDelete(Id_t evengroup_id)

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

13.3.3. EventgroupFlagsSet

void EventgroupFlagsSet(Id_t eventgroup_id, U8_t mask)

Descrip.:	Sets the masked flags in the eventgroup register. if mask = (EVENTGROUP_FLAG_MASK_3 |EVENTGROUP_FLAG_MASK_7), 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_OK`: if the eventgroup was successfully deleted. `OS_ERROR`: if the eventgroup could not be found.

13.3.4. EventgroupFlagsClear

EventgroupFlagsClear(Id_t evengroup_id, U8_t mask)

Descrip.:	Clears the masked flags in the eventgroup register. if mask = (EVENTGROUP_FLAG_MASK_0 |EVENTGROUP_FLAG_MASK_4), 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_OK`: if the eventgroup was successfully deleted. `OS_ERROR`: if the eventgroup could not be found.

13.3.5. EventgroupFlagsGet

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.
Return:	`(U8_t)` Eventgroup flags.

14. Ringbuffers

14.1. *Ringbuffer* API

14.1.1. RingbufCreate

RingbufCreate(U32_t size)

Descrip.: Creates a ring-buffer of given size. Note that the width of element in the buffer is defined by RingbufBase_t (U8_t by default).
Input: `(U32_t) size`: Ring-buffer size in elements.
Return: `(Id_t)` Ring-buffer ID
`INVALID_ID`: if an error occurred during creation.
`Other`: valid ID if the ring-buffer was created.

14.1.2.

RingbufDelete (Id_t ringbuf_id)

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

14.1.3. RingbufWrite

RingbufWrite(Id_t ringbuf_id , RingbufBase_t data, U32_t length)

Descrip.: Writes a given number of data elements to the ring-buffer. The of elements actually written is returned.
Input: `(Id_t) ringbuf_id`: Ring-buffer ID.
`(RingbufBase_t) data`: Array of data elements.
`(U32_t *) length`: Length of data in elements.
Return: `(OsResult_t)` sys call result
`OS_OK`: if data was written.
`OS_FAIL`: if the buffer is empty.
`OS_LOCKED`: if the ringbuffer is already locked for writing.
`OS_ERROR`: if the ringbuffer could not be found.

14.1.4. RingbufRead

RingbufRead(Id_t ringbuf_id, RingbufBase_t data, U32_t amount)

Descrip.: Reads a given number of data elements from the ring-buffer. The of elements actually read is returned. Read data is copied to the target array. The target array has to comply with the pre-conditions below.
Input: `(Id_t) ringbuf_id`: Ring-buffer ID.
`(RingbufBase_t) target`: Target data array. Pre-condition: target[0] = 0xFE, target[amount-1] = 0xEF
`(U32_t *) amount`: Amount of data to read in elements.
Return: `(OsResult_t)` sys call result
`OS_OK`: if data was read.
`OS_LOCKED`: if the ringbuffer is already locked for reading.
`OS_FAIL`: if the buffer is empty.
`OS_OUT_OF_BOUNDS`: if the target array was not compliant.

`OS_ERROR`: if the ringbuffer could not be found.

14.1.5. RingbufDump

RingbufDump(Id_t ringbuf_id, RingbufBase_t target)

Descrip.:	Dumps all data present in the ring-buffer in the target array. number of elements actually read is returned. The target array has to comply the pre-conditions stated below.
Input:	`(Id_t) ringbuf_id`: Ring-buffer ID.
	`(RingbufBase_t) target`: Target data array. Pre-condition: target[0] = 0xFE, target[ring-buffer size-1] = 0xEF
Return:	`(U32_t)` Actual amount of elements read from the ring-buffer. 0 if

14.1.6. RingbufFlush

RingbufFlush(Id_t ringbuf_id)

Descrip.:	Resets the ring-buffer's read and write locations as well as 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_OK`: if the ring-buffer was flushed.
	`OS_LOCKED`: if the ring-buffer is locked for reading or writing.
	`OS_ERROR`: if the ring-buffer could not be found.

14.1.7. RingbufSearch

RingbufSearch(Id_t ringbuf_id, RingbufBase_t query, U32_t query_length)

Descrip.:	Searched the ring-buffer for the given query of given length. 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.

14.1.8. RingbufDataCountGet

RingbufDataCountGet(Id_t ringbuf_id)

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

14.1.9. RingbufDataSpaceGet

RingbufDataSpaceGet(Id_t ringbuf_id)

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

15. Logger

15.1. 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.

15.2. *LOG* API

15.2.1. LOG_ERROR_NEWLINE

LOG_ERROR_NEWLINE(message, ...)

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

15.2.2. LOG_ERROR_APPEND

LOG_ERROR_APPEND(message, ...)

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

15.2.3. LOG_DEBUG_NEWLINE

LOG_DEBUG_NEWLINE(message, ...)

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

15.2.4. LOG_DEBUG_APPEND

LOG_DEBUG_APPEND(message, ...)

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

16. Conversions

16.1. 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 in this category are prefix with *Convert*.

16.2. *Convert* API

16.2.1. ConvertUsToMs

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.

16.2.2. ConvertMsToUs

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.

16.2.3. ConvertResultToString

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.
Return: `(U8_t)` number of characters (excluding null
`0`: Operation failed.
`Other`: Valid number of characters.

16.2.4. ConvertIntToString

ConvertIntToString(U32_t integer, char out_int_str)

Descrip.: Converts a integer value to a null-terminated string.
Input: `(U32_t) integer`: Integer value.
Return: `(U8_t)` number of characters (excluding null
`0`: Operation failed.
`Other`: Valid number of characters.

16.2.5. ConvertIntToBytes

ConvertIntToBytes(U32_t integer, U8_t num_bytes, U8_t out_bytes)

Descrip.: Convert an integer value into separate bytes. This allows easy splitting of for instance 32-bits value into 4 bytes. 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.

Return: `(U8_t)` result.
`0`: Number of bytes has an illegal value i.e. 0 or >4.
`1`: Operation successful.

16.2.6. ConvertOsVersionToString

ConvertOsVersionToString(OsVer_t os_version, char out_os_version_str)

Descrip.: Converts OS version to a null-terminated string in the following V<M>.<m>.<u> Major, minor, update.

Input: `(OsVer_t) os_version`: OS Version to convert.

Return: `(U8_t)` number of characters (excluding null
`0`: Operation failed.
`Other`: Valid number of characters.

17. Shell

17.1. 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's API.

17.2. Commands

17.2.1. *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 17.1: 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.

17.2.2. *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.

17.2.3. *tstat*

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

17.2.4. *run*

The specified task is scheduled to run.

Table 17.2: 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.

17.3. *Shell* API

17.3.1. ShellCommandRegister

ShellCommandRegister(struct ShellCommand command)

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

17.3.2. ShellReplyInvalidArgs

ShellReplyInvalidArgs(char command)

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

17.3.3. ShellPut

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.

17.3.4. ShellPutRaw

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.

17.3.5. ShellPutRawNewline

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.