Prior RTOS V 0.3.X

User Manual

not complete

D. van de Spijker

Table of Contents

List of Figures

License and distribution

Prior RTOS is distributed under the MIT License.

The MIT License (MIT)

Abbreviations and definitions

This list contains the abbreviations and definitions used throughout this document along with a brief description.

OS	*Operating System*
ITC	*Inter-Task Communication: The communication between tasks through tools provides by an OS.*
RTOS	*Real-Time Operating System: Operating System with emphasis on the Real-Time aspects of an application ensuring that timing constraints are met.*
RAM	*Random Access Memory*
GUI	*Graphical User Interface*
SPI	*Serial Peripheral Interface (developed by Motorola)*
UART	*Universal Asynchronous Receiver/Transmitter*
I^2C	*Inter IC Communication*
ISR	*Interrupt Service Routine*
ASR	*Asynchronous Signal Routine*
HAL	*Hardware Abstraction Layer*
GC	*Garbage Collector: A set of algorithms that typically trace, mark and free unused allocated memory.*

Introduction

What is Prior?

Prior is an embedded Real-Time Operating System kernel with a small footprint and high customizability enabling it to run smoothly on even the smallest devices like 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 real-time applications becomes a painless process using the wide variety of intuitive API functions.

Core principles

The Prior kernel started out as a learning project to get more familiar with operating systems and design techniques. The kernel was developed with emphasis on customizability, priority and intuitiveness while keeping real-time concepts in mind.

Customizability

The idea is to let the software developer decide what modules to use and which ones are rudimental for a particular application. On the contrary Prior also offers a wide variety of Inter-Task Communication (ITC) modules, Memory Management (MM), a Garbage Collector (GC) and a Graphical User Interface (GUI) framework.

Priority

As the name suggests Prior is a priority based OS. Tasks are assigned one of four available categories (major levels) to indicate its scheduling and time constraints. Furthermore the task is also assigned a minor priority level ranging from 1 through 5, where 5 is the highest priority within its respective category. The assigned priority level is protected from inversion through detection and priority borrowing.

Intuitiveness

All objects created during run-time are assigned a unique ID by the kernel. This ID is used to reference the object when using the API functions. All IDs are maintained within the same ID space meaning that any ID can be locked or added to a task's event list.

Specifications and Features

- Cooperative scheduler based on weighted FIFO.
- *Pre-emptive scheduler to enable instant response to events.
- Priority system with 4 categories to indicate scheduling/timing constraints and a minor priority level ranging from 1 through 5.
- Inter-Task Communication (ITC) system providing events, pipes, mailboxes, semaphores, mutexes and ringbuffers.
- *32 Asynchronous Signals with priority level ranging from 0 to 15.
- *Queue-able asynchronous signals for each individual task.
- Advanced memory management with *Garbage Collector
- *Integrated Graphical User Interface framework for fast and easy GUI development.
- Integrated Command Shell accessible via UART connection.
- *TaskTrace enables the developer to trace every task, timer, event and action to speed up the debugging process.
- OS Tick overhead: ±1000 CPU clock-cycles (7-10% CPU load on AVR8 @ 16MHz)
- Small footprint kernel with a minimal size of 10kB and 250 bytes of RAM.
- Modular design to tailor the OS to your application.

- Memory scalability per module:

Module	Program Memory (kB)	Data Memory (Bytes)
Core		
Tasks		
Timers		
Memory Management		
Total		
Events		
Pipes		
Semaphores & Mutexes (SMX)		
Mailboxes		
Ringbuffers		
Entities		
Signals		
Shell		
TaskTrace		

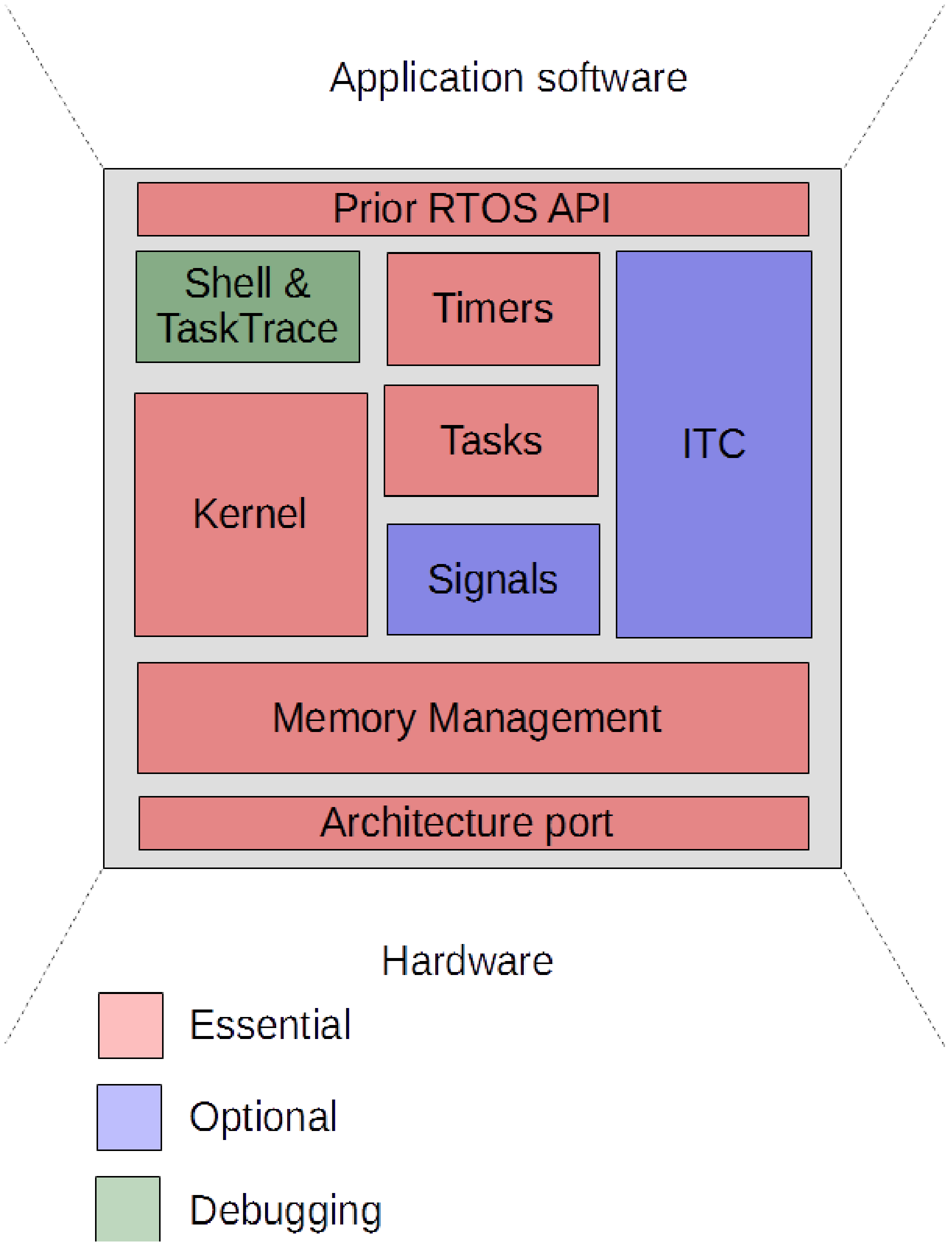
- Data memory scalability per unit:

Unit name	*Data memory (Bytes)*
Task Control Block	15
Timer Control Block	12
Memory Pool Control Block	5 + (sizeof(MemWidth_t)*3)
Event Control Block	4
Pipe Control Block	5+size
SMX Control Block	10
Ringbuffer Control Block	
Mailbox Control Block	
Entity Control Block	

* Still in development

High Level Architecture

Figure X shows the high level architecture of Prior RTOS. The blocks colored red are essential to the kernel and can therefore not be excluded. When a bare-minimum configuration is chosen i.e. none of the additional modules are enables, the OS will still provide kernel control, task management, software timers and memory management (without garbage collector). The optional modules are highlighted in blue and mainly consist of Inter-Task Communication modules. The optional modules include *event, pipes, ringbuffers, mailboxes, semaphores, mutexes and entities.*
The block highlighted in green contain the modules for debugging purposes. Prior RTOS provides a shell accessible via UART (See section **Prior Shell**) and a system called TaskTrace (see section **TaskTrace**) that interacts with a GUI giving the developer an overview of all kernel objects and more.



OS definitions

Files: Prior_types.h

Types

U8_T_t	*Unsigned 8-bits Integer*
S8_t	*Signed 8-bits Integer*
U16_T_t	*Unsigned 16-bits Integer*
S16_t	*Signed 16-bits Integer*
U32_T_t	*Unsigned 32-bits Integer*
S32_t	*Signed 32-bits Integer*
U64_t	*Unsigned 64-bits Integer*
S64_t	*Signed 64-bits Integer*

ID_t	*Identity type, used for all kernel objects, 16 bits*
Stat_t	*Status type, used to inform caller of performed action*
Task_t	*Task type, a pointer to the task code*
TskCat_t	*Task Category type (enumerate), contains the different task categories (cat_OS, cat_realtime, cat_high, cat_medium, cat_low)*
Prio_t	*Priority type, represents the minor priority level ranging from 1-5 for task and from 1-15 for signals.*
TskState_t	*Task State type (enumerate), contains the possible states of a task (disabled, suspended, blocked, idle, dormant, active_L, active_H, scheduled, running, critical)*

TmrState_t	*Timer State type (enumerate), contains the possible states of a timer (stopped, running, waiting)*
MemWidth_t	*Memory Width type, represents the width of the memory architecture.*
StackWidth_t	
MailboxWidth_t	
RingWidth_t	*Ringbuffer Width type, defines the width of each ringbuffer in the ringbuffer module. Default: U8_T_t*
OSVer_t	*OS Version type, 16-bits; 0x0031 = V 0.3.1*

Macros

INV_ID	*Invalid ID (0xFFFF)*
TMRP_ON	*Timer Parameter ON (0x01)*
TMRP_AR	*Timer Parameter Auto Reset (0x02)*
TMRP_P	*Timer Parameter Permanent* (0x04)
PRIOR_COOP	*Prior Cooperative Scheduler*

Prior Configuration file

File: Prior_config.h

This chapter contains all the settings that are available within Prior RTOS. Prior comes with this wide variety of settings to ensure that the OS can be tailored to your application, making the use of an OS such as Prior also possible on small-RAM devices such as the PIC or AVR8.

Keep in mind that almost every setting that you change will have influence on the size, speed and RAM use of the OS.

ENABLE_: {disabled=0; enabled>=1}

Global settings

CFG_F_CPU	: CPU frequency in Hz e.g. 20000000UL (20MHz)
CFG_F_OS	: Operating System frequency in Hz e.g. (U16_T) 1000
CFG_SCHEDULER	: Scheduler type {PRIOR_COOP; *PRIOR_PREM} PRIOR_COOP: Prior RTOS Cooperative scheduler.
CFG_RTTC	: Real-Time Task Constraint in milliseconds e.g. 10 ms. The RTTC defines the maximum amount of time allowed between activation and the actual execution of tasks in the category *cat_realtime* (see Task Priority system page XX).

Memory Management settings

CFG_OS_HEAPSIZE : Operating System Heap size in bytes.

CFG_N_POOLS : Maximum number of pools available.

CFG_ENABLE_POOLPROTECTION: Enables pool protection i.e. padding and CRC

CFG_N_STACKS : Number of task stacks available. More stack results in more tasks executing in pseudo-parallel.

CFG_STD_STKDEPTH : Standard Stack size for each running task.

Module settings

CFG_ENABLE_MAIL : Enables Mailbox API

CFG_MAILBOXSIZE : Defines the size of each mailbox

CFG_ENABLE_SMX : Enable Semaphore and Mutex API

CFG_ENABLE_EVENTGROUPS : Enables Event Group API

CFG_ENABLE_PIPES : Enable Pipe API

CFG_ENABLE_RINGBUFS : Enable Ringbuffer API

CFG_ENABLE_ENTITIES : Enable Entity API

CFG_ENABLE_SIGNALS : Enable Asynchronous Signals and its API

CFG_ENABLE_TASKTRACE : Enables TaskTrace (UART & Shell have to be enabled)

CFG_ENABLE_SHELL : Enables Shell commands via UART (UART has to be enabled)

Additional settings

CFG_ENABLE_WATCHDOG	: Enable Watchdog timer
CFG_ENABLE_RUNTIMECALC	: Enable task runtime calculations.
CFG_ENABLE_CPULOADCALC	: Enable CPU load calculations.
CFG_ENABLE_TASKNAMES	: Enable generic names for each task. This option is mainly used for debugging.
CFG_TASKNAME_LENGTH	: Maximum amount of characters a task name may contain.

Peripheral settings

CFG_ENABLE_GUI	: Enable Prior GUI Framework
CFG_ENABLE_SPI	: Enables SPI and its API
CFG_ENABLE_SDCARD	: Enables the use of the SD card driver and its API
CFG_ENABLE_PWM	: Enables the soft-PWM driver and its API
CFG_ENABLE_UART	: Enables hardware UART if port is provided, soft-UART if it is not.
CFG_UART_BAUDRATE	: Sets the baud rate for the hardware UART
CFG_SHELLPW	: Shell password for root access

Architecture Port

Files: Port.h – Port.c

The architecture port files provide the Hardware Abstraction Layer (HAL) used by the kernel to access hardware e.g. timers, interrupt controllers.

port_OSTimerInit

port_OSTimerStart

port_OSTimerStop

port_OSTimerReset

port_OSTimerCountGet

port_OSTickStart

port_OSTickStop

port_GlobalInterruptsDisable

port_GlobalInterruptsEnable

port_OSInterruptDisable

port_OSInterruptEnable

port_WatchdogDisable

port_WatchdogEnable

Deployment

The way of deploying Prior RTOS is focused on being straight-forward and fast, without the hassle of having to search the source folder for the right header files. Instead, Prior uses a single header file that automatically includes the rest of the headers based on the configuration file.

Pre-deployment requirements

- C99 compatible compiler

Deploying Prior RTOS

Step 0: Copy the source file of the selected port (PriorRTOS\Ports\<selectedport>\Port.c) to PriorRTOS\Kernel. Then copy the header file of the selected port (PriorRTOS\Ports\<selectedport>\Port.h) to PriorRTOS\Include.

Step 1: Include the PriorRTOS folder in the solution.

Step 2: Open the configuration file (PriorRTOS\Include\Prior_config.h) to change the configuration settings. Refer to **Prior Configuration File** section for more information on these settings.

Step 3: Remove the files of the modules that have not been enabled in the configuration file from the PriorRTOS\Kernel folder.

Step 4: Include the Prior RTOS main header in the main file of the solution (PriorRTOS\Include\Prior_RTOS.h) .

(Step 5 :) Include Prior_example.c

Task Priority System

Prior RTOS uses an easy-to-understand priority system to help the kernel plan what task is executed at a certain point in time. To help you as a programmer, 4 categories were implemented that indicate the major priority level of each task; realtime, high, medium, low (Table X for full description). Each priority category has 5 minor priority levels ranging from 1 through 5, where 5 is the highest priority within its respective category.

Category(Major)	*Description*	*Example*
OS (4)	Restricted category only to be used by the kernel.	N/A
realtime (3)	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 the RTTC setting.	Signal sampling task
high (2)	High priority tasks are still very important to their system, but missing a deadline would not result in critical errors. Tasks in the category *high* are guaranteed to be scheduled after one scheduler cycle.	State Machine task
medium (1)	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. Tasks in the category *medium* are guaranteed to be scheduled after two scheduler cycles.	Calculation task
low (0)	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. Tasks in the category *low* are scheduled	Heartbeat task

OS Objects and the API

File: Prior_RTOS.h

In this section all OS objects and their respective APIs are discussed in detail. All of the API function prototypes with a brief description can be found in the Prior RTOS header file.

Kernel Control

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 general API functions have the prefix **os_**

os_Init

```
Stat_t os_Init (      void
               )
```

This function initiates the Prior kernel, it should be called before calling any other API function.

Parameters:

N/A

Return:

(Stat_t)	Operation status: e_ok if initiation was successful. e_fail if one of the non-essential modules was not initiated successfully. In this case the kernel is still operational. However it is recommended to call **os_Reset** or reset the host processor. e_error if one of the core modules was not initiated correctly. **os_Reset** has to be called or the host processor has to be reset.

os_Start

```
void os_Start (      Task_t handler
              )
```

Starts the Prior kernel including the OS-timer at the set frequency, CFG_F_OS by default. The core will begin executing given task. If the task handler was not found in of the lists, the idle task will be scheduled.
NOTE: The software will not return from this function, code below this point within the same scope will not be executed.

Parameters:

(Task_t) handler	Task handler address of starting task. NULL if starting task is not specified, Idle task will be scheduled.

Return:

N/A

Example:

```
os_Start(SysInit);
```

os_FrequencySet

```
Stat_t os_FrequencySet (        U16_T frequency
                       )
```

Sets a new OS frequency. Timer pre-scalar and timer overflow value will be recalculated if necessary. The kernel will continue to operate on this frequency immediately after this function is called. Resetting the frequency to its base can be done by passing F_OS as the parameter.

Parameters:

(U16_T) frequency	New OS frequency in Hz

Return:

(Stat_t)	Operation status: e_ok if operation was successful. e_fail if the new frequency was higher than 5250.

Example:

```
os_FrequencySet (2500);
```

os_FrequencyGet

```
U16_T os_FrequencyGet (        void
                      )
```

Returns the current OS frequency in Hz.

Parameters:

N/A

Return:

(U16_T)	Current OS frequency in Hz.

Example:

```
U16_T f_os =
os_FrequencyGet();
```

os_VersionGet

```
OSVer_t os_VersionGet (      void
                      )
```

Returns the current OS version in the following format: 0x0032 = V 0.3.1

Parameters:

N/A

Return:

(OSVer_t) Current OS Version

Example:

```
OSVer_t cur_osver = os_VersionGet();
```

os_RuntimeGet

```
Stat_t os_RuntimeGet (      U32_T* target
                     )
```

Copies the current OS runtime to the target array. Target has to have at least 2 elements filled with 0x00000000;

Parameters:

(U32_T)* target Pre*: t*arget array of at least 2 elements initiated at 0x00000000.
Post: target [0] = hours, target [1] = microseconds

Return:

(Stat_t) Operation status:
e_ok if operation was successful.

e_error if the target array did NOT comply with the requirements stated in the description.

Example:

```
U32_T runtime[2] = {};
Stat_t status = os_RuntimeGet(&runtime);
if (status == e_ok)
{
  //read/process runtime here
}
```

os_TickTGet

```
U32_T os_TickTGet (        void
                 )
```

Returns the current OS tick period in microseconds.

Parameters:

N/A

Return:

(U32_T)	Tick period in microseconds

os_CritSectEnter

```
void os_CritSectEnter (      void
                     )
```

Requests the kernel to lock the scheduler and disable interrupts, protecting the code following this operation from interruptions. The request is granted if and only if there are no real-time tasks with higher priority than the calling task scheduled.
NOTE 1: This function should ONLY be used in critical sections like CRC generation/validation or sections that require precise timing.
NOTE 2: The task calling **os_CritSectEnter** will be dispatched if it does not call **os_CritSectExit** within the specified time-window,

Parameters:

N/A

Return:

(Stat_t)	Operation status: e_ok if operation was successful. e_fail if the request was not granted.

Example:

```
Stat_t status = os_CritSectEnter();
if (status == e_ok)
{
  //execute critical code here
  //..
  //..
  os_CritSectExit();
}
```

os_CritSectExit

```
Stat_t os_CritSectExit (      void
                         )
```

Unlocks the scheduler and enables interrupts. The code following this statement is no longer protected again interruptions.
NOTE: This function has to be called after calling **os_CritSectEnter.**

Parameters:

N/A

Return:

(Stat_t)	Operation status: e_ok if operation was successful. e_fail if the scheduler and interrupts were enabled before calling this function.

Example: See Example of **os_CritSectEnter**

os_ISREnter

void **os_ISREnter (** *void*
)

Signals the kernel that another Interrupt Service Routine is executed. In this protected section, no dispatching will take place. It is recommended to keep ISRs concise to minimize overhead.
NOTE: This function has to be followed by **os_ISRExit.**

Parameters:

N/A

Return:

N/A

Example:

```
os_ISREnter();
//ISR code here
//..
//..
os_ISRExit();
```

os_ISRExit

void **os_ISREnter (** *void*
)

Signals the kernel that another Interrupt Service Routine is executed. In this protected section, no dispatching will take place. It is recommended to keep ISRs concise to minimize overhead.
NOTE: This function has to be followed by **os_ISRExit.**

Parameters:

N/A

Return:

N/A

Example: See Example of **os_ISREnter**

os_SchedulerLock

void **os_SchedulerLock (** *void*
)

Signals the kernel to lock the scheduler. Tasks can still be interrupted and dispatched if a task with a higher priority is scheduled before calling this function.
NOTE: This function has to be followed by **os_SchedulerUnlock**

Parameters:

N/A

Return:

N/A

Example:

```
os_SchedulerLock();
//code here
//..
//..
os_SchedulerUnlock();
```

os_SchedulerUnlock

void **os_SchedulerUnlock (** *void*
)

Signals the kernel to unlock the scheduler.
NOTE: This function is always called in combination with **os_SchedulerLock**

Parameters:

N/A

Return:

N/A

Example: See Example **os_SchedulerLock**

Memory Management

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 CFG_OS_HEAPSIZE 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 heap can be split into N pools, where N is defined by CFG_N_POOLS. Each pool's size is configured upon creation. Furthermore, allocating memory using **mm_DynAlloc** or **mm_StcAlloc** has the following advantages:

- It can be protected by padding and a pool-checksum. If an execution 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.
- It is guaranteed to be consecutive.
- It can be managed by pool operations allowing for sorting, defragmenting, formatting and moving to keep the pool in an optimal condition.
- It can be managed by the Garbage Collector to avoid memory leaks and dangling pointers.

Memory heap and pool operations

mm_OSHeapFreespaceGet

mm_PoolCreate

```
ID_t mm_PoolCreate (        U32_T pool_size
                   )
```

Creates a pool of size *pool_size* in bytes located in the OS heap. After creating a pool, allocations can be made inside this pool.

Parameters:

(U32_T) pool_size Size of the new pool in bytes.

Return:

(ID_t) Returns the pool ID if creation was successful.
Return INV_ID (0xFFFF) if there is not enough space available on the OS Heap at the time of creation OR if the maximum number of available pools (CFG_N_POOLS) was reached.

Example:

```
ID_t new_pool = mm_PoolCreate(100); //Creates a new pool
if(new_pool != INV_ID)
{
   //Allocations here
   mm_Alloc(…);
}
```

mm_PoolDelete

```
Stat_t mm_PoolDelete (        ID_t pool_ID
                     )
```

Zeroes all content remaining of the selected pool and returns this memory to the OS Heap. The used control block is now available for the creation of a new pool.

Parameters:

(ID_t) pool_ID   Pool ID

Return:

(Stat_t)   Operation status:
e_ok if operation was successful.

e_error if the pool ID does not exist or if the control block is currently not in use.

Example:

```
ID_t tmp_pool = mm_PoolCreate(100); //Creates a new pool
if(tmp_pool != INV_ID)
{
   //Allocations here
   mm_Alloc(..);

   //Use allocated memory here
   //...

   mm_Free(..);

   if(mm_PoolDelete(tmp_pool) != e_ok) //Attempt to delete
   {
     //error handling here
   }
}
```

mm_PoolFormat

```
Stat_t mm_PoolFormat (       ID_t pool_ID
                     )
```

Zeroes all content of the selected pool. The pool's health will be restored to 100% and all the initial space is again available for allocations.

Parameters:

(ID_t) pool_ID   Pool ID

Return:

(Stat_t)   Operation status:
e_ok if operation was successful.

e_error if the pool ID does not exist or if the control block is currently not in use.

Example:

```
ID_t tmp_pool = mm_PoolCreate(100); //Creates a new pool
if(tmp_pool != INV_ID)
{
   //Allocations here
   mm_Alloc(..);

   //Use allocated memory here
   //...


   if(mm_PoolFormat(tmp_pool) != e_ok) //Attempt to format
   {
     //error handling here
   }
}
```

mm_PoolDefrag

```
Stat_t mm_PoolDefrag (      ID_t pool_ID
                    )
```

Defragments the selected pool closing gaps between allocated blocks resulting in more continuous memory.

Parameters:

(ID_t) pool_ID   Pool ID

Return:

(Stat_t)   Operation status:
e_ok if operation was successful.
e_error if the pool ID does not exist or if the control block is currently not in use.

Example:

mm_PoolMove

mm_PoolSort

mm_PoolHealthGet

mm_PoolFreespaceGet

```
U32_T mm_PoolFreespaceGet (        ID_t pool_ID
                         )
```

Returns the amount of free space available in the selected pool. If **mm_PoolFreespaceGet** returns values above 0 yet **mm_Alloc** consequently returns NULL then defragmenting the pool is recommended using **mm_PoolDefrag**.

Parameters:

(ID_t) pool_ID	Pool ID

Return:

(U32_T)	Free space left in bytes

Example:

N/A

Garbage Collector options

mm_CollectorEnable

mm_CollectorDisable

Static memory allocation

mm_StcAlloc

Dynamic memory allocation

mm_DynAlloc

```
void* mm_DynAlloc (      ID_t pool_ID,
                       U32_T      size
                )
```

Allocates a specified amount of continuous memory in the selected pool. **mm_Alloc** returns a pointer to the allocated memory if the operation succeeded.

Parameters:

(ID_t) pool_ID	Pool ID*,* the memory will be allocated in this pool
(U32_T) size	Requested size

Return

(void)*	Pointer to memory if successful. NULL if the pool has no continuous memory available of the specified size.

Example:

```
U32_T* my_alloc = mm_DynAlloc(my_pool,sizeof(U32_T));

*my_alloc = 0xDEADBEEF;

//Manipulate data here

mm_ReAlloc(my_pool, my_alloc, sizeof(U64));

*my_alloc = 0xDEADBEEFD00FF00D;

//Manipulate data here

mm_Free(my_pool, my_newalloc);
```

mm_ReAlloc

```
Stat_t mm_ReAlloc (      ID_t     pool_ID,
                         void*    ptr
                         U32_T    new_size
                 )
```

Re-Allocates a specified amount of continuous memory in the selected pool. This pool must be the same pool the original allocation. **mm_ReAlloc** changes the ptr argument to the newly allocated address.

Parameters:

(ID_t) pool_ID	Pool ID*,* the memory will be allocated in this pool
(void)* ptr	Pointer to current allocation. Changed after successful re-allocation.
(U32_T) new_size	New requested size

Return

(Stat_t)	Operation status: e_ok if re-allocation was successful. e_fail if the requested new size could not be granted.

Example: See Example **mm_Alloc**

mm_Free

```
Stat_t mm_Free   (      ID_t     pool_ID,
                         void*    ptr
                  )
```

Frees allocated memory, returning the free space to its respective pool. Freed memory is automatically zeroed.

Parameters:

(ID_t) pool_ID — Pool ID*,* memory to be freed resides in this pool.

(void)* ptr — Pointer to allocation.

Return

(Stat_t) — Operation status:
e_ok if memory was freed.

e_fail if the pointer argument was invalid (either NULL or not within the OS heap space).

Example: See Example **mm_Alloc**

Tasks

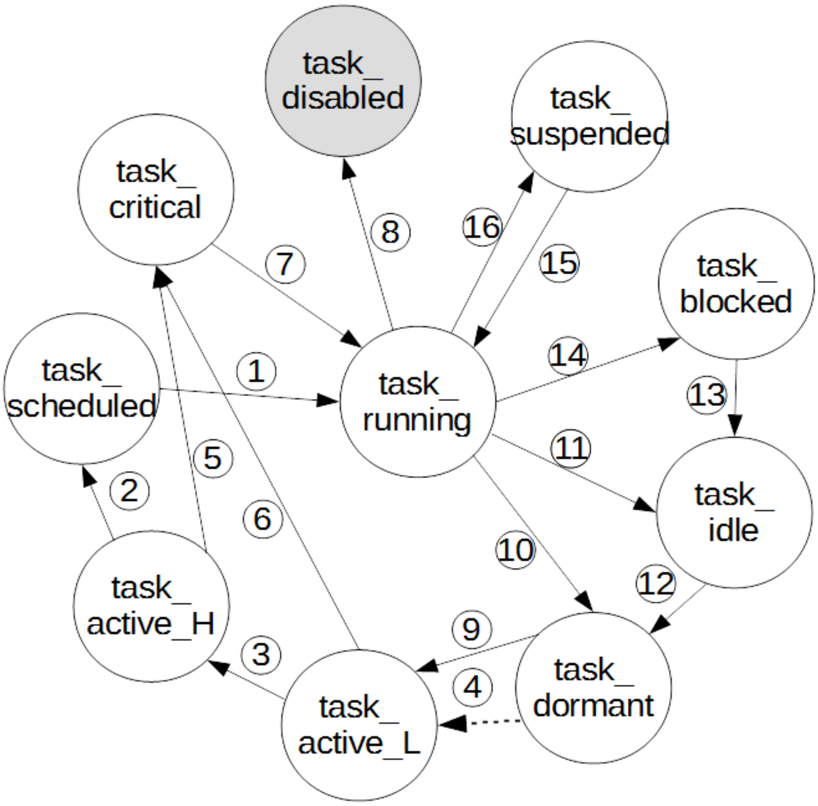
Tasks are one of the essential objects within Prior. They enable the programmer to split their code into smaller parts. Within these smaller parts code is sequential as usual, but the smaller parts are given execution time by the OS kernel. Of course there are ways to influence the task-flow like synchronization, but in the end the scheduler decides.
All task API functions have the prefix `task_`

Task Control Block

Creating a new task consumes 15 bytes of RAM until the task is deleted. A single Task Control Block (TCB) contains the following attributes that can be accessed by the user through API functions:

Attribute	Get	Set	Reset
Handler address	**Yes**	**Yes**	**No**
Priority level	**Yes**	**Yes**	**No**
Category	**Yes**	**Yes**	**No**
Event list	**Yes**	**Yes**	**Yes**
Mailbox address	**Yes**	**Yes**	**Yes**
Task state	**Yes**	**No**	**No**
Average run-time	**Yes**	**No**	**Yes**
Argument	**Yes**	**Yes**	**No**

Task States



1. scheduler action; first in execution queue is swapped in during task switch.
2. scheduler action; task is placed in execution queue.
3. scheduler action; task is promoted to active_H level.
4. event
5. active time > RTTC (only cat_realtime tasks)
6. active time > RTTC (only cat_realtime tasks)
7. scheduler action; critical task is executed instead of first in execution queue.
8. kernel action; task caused memory overflow.
9. task_Wake is called before a event has been generated.
10. task_Wait(NULL, ...) (V0.3 is called by the running task.
11. task_Yield was called or the task has finished executing.
12.task_Wait(task x, ...) (V0.3) is called by running task y. 13.
14.
15. task_Resume(task x) is called by running task y.
16. task_Suspend(NULL) or task_Suspend(task x) is called by task x or task **y.**

Task Event List *V 0.3.1 and successive versions*

Tasks have to be in an activate state in order to be scheduled. For a task to be transitioned to this state, an external source is required. This external source can either be a **signal**, a manual activation by another task using **task_Wake** or an **event**. Event are generated by other kernel objects, and in most cases they can generate multiple type of Event. A task can add a particular event to its Event List using **task_Wait.**
The event-generating object is referenced by its ID, the event can be selected using a 16-bits parameter. Bit 15 may be set high (1) to indicate that the event is permanent, if bit 15 is left at 0 the event will be cleared from the task's Event List after handling. This particular task is then sensitive to this event and will be transitioned to an active state automatically when this event is generated.

task_Create

```
ID_t task_Create  (     Task_t handler,
                        task_cat_t task_category,
                        PRIO_t task_priority
                  )
```

Creates a Task Control Block for the task handler, enabling it to be scheduled by the kernel. Tasks can dynamically be created and can be scheduled as soon as the next OS tick. _Create may be called by other task or from within ISRs.

Parameters:

(Task_t) handler	Address of the task handler.
(tskcat_t) task_category	Priority category; low, medium, high or realtime.
(PRIO_t) task_priority	Task minor priority level; 1-5.

Return:

(*ID_t*)	Returns the 16-bits ID attached to the TCB by the kernel if the task was created successfully. INV_ID (0xFFFF) if the was not created.

Example:

```
ID_t id1 = task_Create(ledblink, medium, 2);

void ledblink(void* parameter)
{
      PORTB ^= (1 << PINB0);
}
```

task_Delete

```
stat_t task_Delete(     Task_t handler
                      )
```

Deletes the TCB belonging to a task. The task will virtually not exist anymore for the kernel. Tasks can dynamically be deleted, garbage collection happens after the task is completed. _Delete may be called by other task or from within ISRs.

Parameters:

*(Task_t)*handler	Address of the task handler.

Return:

(*Stat_t*)	Returns e_ok if the operation was successful, e_fail if the operation has failed.

Example:

```
task_Delete(ledblink);
```

task_Wait

```
Stat_t task_Wait        (            Task_t handler,
                                     ID_t   object_ID,
                                     U8_T     mask
                         )
```

Sets the trigger object for the given task. Whenever the given object generates an event, the task will be transitioned to the active state and become ready for scheduling. When not running the task's state is set to *task_dormant*, and will reside in the waiting list. Examples of these events are a timer overflow, an event flag set or a ringbuffer-write. The mask can be used to select multiple event flags that will each trigger the task when set.

NOTE: **task_Wait** cannot activate tasks in the *idle, suspended, blocked* or *disabled* state.

Parameters:

(Task_t) handler	Task entry point; NULL to address the currently running task
(ID_t) object_ID	Wait object ID
(U8_T) mask	Event flag mask

Return: *N/A*

Example:

```
ID_t id1 = task_Create(ledblink, cat_medium, 5);

ID_t ex_event = event_Create();


void ledblink(void* parameter)
{
      PORTB ^= (1 << PINB0);
      task_Wait(NULL,ex_event,0x06);
      //ledblink can now be triggered by flag
      //# 2 and 3 of ex_event
}
```

task_EventHandle

added V 0.3.1 and successive versions

task_Wake

```
Stat_t task_Wake (      Task_t handler,
                        void*  parameter
                  )
```

Wakes a specific task from the *task_idle* or *task_dormant* state setting the new state to *task_active_L.* A parameter pointer can be passed to this specific task, which can then be de-referenced on execution. (See also **task_ParameterGet** and **task_ParameterSet**)
NOTE: **task_Wake** cannot activate tasks in the *suspended, blocked* or *disabled* state.

Parameters:

(Task_t) handler	Address of the task handler.
(void)* parameter	Parameter passed to the receiving handler, receiving handler is responsible for dereferencing to the correct type.

Return:

(*Stat_t*)	Status of the operation. e_fail when task is in one of these states: *task_blocked, task_suspended* or *task_disabled.* e_error when task control block with given handler does not exist. e_ok when operation was successful.

Example:

```
U32_T data = 0xFFFFEEEE;
task_Wake(ledblink,&data);

void ledblink(void* parameter)
{
      U32_T data = (U32_T*) parameter;
      //process data here
      // ..
      // ..
      PORTB ^= (1 << PINB0);
}
```

task_Suspend

```
Stat_t task_Suspend(    Task_t handler
                   )
```

Yields the given task transitioning it to the *suspended* state. The task will NOT be triggered by other objects or **task_Wake** calls while it is in this state. Calling **task_Resume** is the only way activate the given task.

Parameters:

(Task_t) handler — Address of the task handler; NULL to reference the currently running task.

Return:

(*Stat_t*) — Status of the operation.

e_fail when task is in one of these states: *task_blocked, task_suspended* or *task_disabled.*
e_error when task control block with given handler does not exist.
e_ok when operation was successful.

Example:

```
void ledblink(void* parameter)
{

      PORTB ^= (1 << PINB0);

      task_Suspend(NULL);
      //Suspends ledblink until
      //task_Resume is called by a another task.
}
```

task_SuspendAll

```
Stat_t task_SuspendAll(
                       )
```

Yields all tasks in all lists transitioning them to the *suspended* state. These tasks will NOT be triggered by other objects or **task_Wake** calls while they are in this state. Calling **task_Resume** is the only way activate a given task.

Parameters:

N/A

Return:

(*Stat_t*)	Status of the operation. e_fail when task is in one of these states: *task_blocked, task_suspended* or *task_disabled.* e_error when task control block with given handler does not exist. e_ok when operation was successful.

Example:

```
void ledblink(void* parameter)
{

      PORTB ^= (1 << PINB0);

      task_SuspendAll();
      //Suspends ledblink until
      //task_Resume is called by a another task.
}
```

task_Resume

task_ResumeAll

task_Sleep

task_Delay

task_Yield

task_GenericNameSet

task_GenericNameGet

task_ArgumentSet

task_PrioritySet

```
Stat_t task_PrioritySet  (    Task_t  handler,
                              Prio_t  priority
                          )
```

Assigns a new Minor priority level to the given task, ranging from 1 through 5 (See **Task Priority System** for more information). New priority is used during the next scheduling cycle.

Parameters:	*(Task_t)*	handler	Task handler start address
	(Prio_t)	priority	New task priority

Return:

(*Stat_t*)	Status of the operation. e_fail if the new priority is <1 or >5 e_error when task control block with given handler does not exist. e_ok when operation was successful.

Example:

```
task_PrioritySet(ledblink, 1);
```

task_PriorityGet

```
Prio_t task_PriorityGet ( Task_t handler
                         )
```

Returns the current priority level of the given task. This priority level is a value between 1 and 5. **task_PriorityGet** will return 0 if the handler's TCB was not found.

Parameters: *(Task_t)* handler Task handler start address

Return:

(*Prio_t*) Current task priority

1-5 if operation was successful.
0 when task control block with given handler does not exist.

Example:

```
Prio_t tsk_prio = task_PriorityGet(ledblink);
```

task_CategorySet

task_CategoryGet

task_StateGet

task_RuntimeGet

task_RuntimeReset

Timers

Timer objects are software timers managed by the kernel. Each timer is updated during the OS tick and will generate a event at overflow. All timer API functions have the prefix **`timer_`**

Timer Parameter

Every timer object contains a timer parameter register that is used to configure the timer's 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: Permanent-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 P-bit is 0.

Figure X shows a visual representation of the timer parameter register.

Timer Parameter Register

7	6	5	4	3	2	1	0
It 4 MSB	It 3	It 2	It 1	It 0 LSB	AR	P	ON

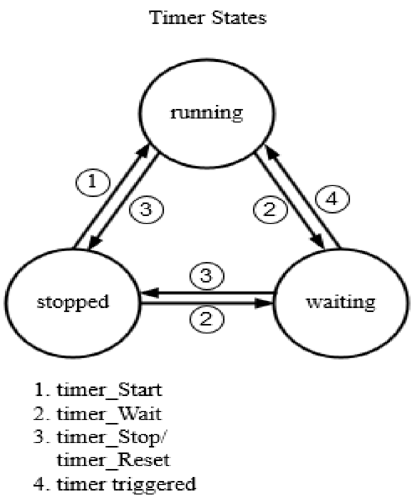
Table X shows the various definitions used to configure the timer parameter.

Definition	**Value**	**Description**
TMRP_ON	0x01	Timer is on by default
TMRP_P	0x02	Timer is permanent
TMRP_AR	0x04	Timer resets automatically on overflow

Timer States

A timer can be in one of three states; *timer_running, timer_waiting, timer_stopped.* Figure X shows the state diagram of these states and their respective transitions.

timer_stopped: The timer is in an inactive state, this is the default state after creation.
timer_running: The timer is in an active state where its counter is incremented every OS tick.
timer_waiting: Timer is waiting for a trigger, until triggered timer is inactive.



Timer Events

Timers can generate the following Event:

Event	*Generated when*	*Bit number in parameter*
Start	The timer is started.	0
Stop	The timer is stopped.	1
Reset	The timer is reset.	2
Overflow	The timer overflows.	3
Deleted	The timer is deleted	4

timer_Create

```
ID_t timer_Create   (       U32_T interval,
                            U8_T  parameter
                     )
```

Creates a software timer object with given overflow in µs and parameter. Information about the timer parameter can be found in the **Timer Parameter** section.
NOTE: For timers to work efficiently the interval should be below the OS tick interval.

Parameters:

(U32_T) interval	Desired timer interval in µs
(U8_T) parameter	Timer parameter

Return:

(ID_t)	Timer ID if the timer was created successfully. INV_ID (0xFFFF) if the timer was not created.

Example:

```
ID_t tmr = timer_Create(20000, (0x05 << 3 | TMRP_AR | TMRP_ON);
//Creates a new timer running at 50Hz (20ms interval)
// 0x05 << 3 : This timer will overflow 5 times, after the fifth
// time it is deleted by the kernel => NOT permanent
// TMRP_AR : This timer resets automatically on overflow
// TMRP ON : This timer is ON by default.
```

timer_Delete

timer_Start

timer_Stop

timer_Pause

timer_Reset

timer_StartAll

timer_StopAll

timer_ResetAll

timer_StateGet

timer_TicksGet

timer_IntervalSet

timer_IntervalGet

timer_IterationsSet

timer_IterationsGet

timer_ParameterGet

timer_ParameterSet

Event Groups

Events groups or event registers contain 8 individual flags that can be set, cleared or acquired by any task or ISR.
All event group API functions have the prefix **`eventgrp_`**

Event	*Generated when*	*Bit number in parameter*
Flag Set	Flag n is set	n (0-7) + bit 8 = 1
Flag Cleared	Flag n is cleared	n (0-7) + bit 8 = 0
AND operation on n flags	All selected flags n in the parameter are set.	bit 9 = 1
OR operation on n flags	One of the selected flags n in the parameter is set.	bit 9 = 0
Deleted	The event was deleted	bit 10

eventgrp_Create

eventgrp_Delete

eventgrp_FlagsSet

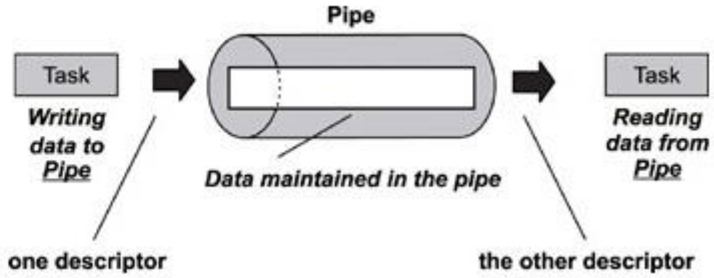
eventgrp_FlagsClear

eventgrp_FlagsGet

eventgrp_Broadcast

Pipes

Pipes are buffers with FIFO semantics designed for Inter-Task Communication. They can hold a specified amount of data which is defined at creation.
All pipe API functions have the prefix **`pipe_`**



Event	*Generated when*	*Bit number in parameter*
Data in	Any amount of data enters the pipe.	0
Data out	Any amount of data is read from the pipe.	1
Pipe full	The pipe is full, no data will be accepted beyond this point.	2
Pipe empty	The pipe is empty.	3
Deleted	The pipe is deleted	4

pipe_Open

pipe_Close

pipe_Write

pipe_Read

pipe_DataLengthGet
pipe_Flush

Semaphores & Mutexes

Semaphores and Mutexes are small kernel objects used to lock entities and other kernel objects. These locks can be used to keep tasks from accessing a (shared) resource.
Prior distinguishes 4 kinds:
- Binary semaphore; not owned by specific task, single flag **(type 1)**
- Counting semaphore; not owned by specific task, multiple flags **(type 2)**
- Mutex; owned by specific task, single flag **(type 3)**
- Recursive mutex; owned by specific task, single flag (allowed to be locked/unlocked multiple times by owner task) **(type 4)**

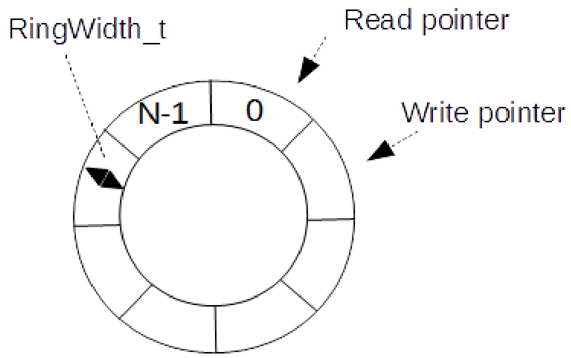
All semaphore and mutex API functions have the prefix **`smx_`**

smx_Create
smx_Broadcast
smx_Delete
smx_Acquire
smx_Release
smx_CountGet
smx_CountReset

Event	*Generated when*	*Bit number in parameter*
Acquired	A semaphore/mutex flag was acquired.	0
Released	A semaphore/mutex flag was released.	1
Count reset	A semaphore counter was reset.	2
		3
Deleted	The semaphore/mutex was deleted.	4

Ring-buffers

Ring-buffers are part of the Inter-Task Communication system and are often used to pass large amounts of data from one task (or ISR) to another. Ring-buffers lend their name to the fact that the buffer's last element is attached to its first element, effectively creating a circle. These buffers based on FIFO semantics manages a read and write pointer that move from one element to the next in a clockwise manner. The write pointer can only propagate to the element before the read pointer and vice versa, ultimately protecting the unread data from being overwritten by newer data. The element width of the ringbuffer module can be set by changing the definition of *RingWidth_t*.



Event	Generated when	Bit number in parameter
Data in	Any amount of data enters the ringbuffer.	0
Data out	Any amount of data is read from the ringbuffer.	1
Ringbuffer full	The ringbuffer is full, data has to be read in order for the ringbuffer to accept new incoming data.	2
Ringbuffer empty	The ringbuffer is empty.	3
Ringbuffer flushed	The ringbuffer was flushed.	4
Deleted	The ringbuffer is deleted	5

ringbuf_Create
ringbuf_Delete
ringbuf_Write
ringbuf_Read
ringbuf_Flush

Mailboxes

A mailbox is an array of width *MailboxWidth_t* and length N, where N is specified upon creation. The array can be written to at every index by other tasks by using **mail_Post,** however, the data can only be read (**mail_Pend**) by its owner. A mailbox may have multiple owners, the maximum amount is specified upon creation. A mailbox owner can be added or removed using **mail_OwnerAdd** and **mail_OwnerRemove** respectively.

Event	*Generated when*	*Bit number in parameter*
Post	Data was posted in the mailbox.	0
Pend	Data was pended from the mailbox.	1
Empty	The mailbox is empty	3
Delete	The mailbox was deleted	4

mail_OwnerAdd

mail_OwnerRemove

mail_Post

mail_Pend

mail_Flush

Asynchronous Signals

sig_RoutineAssign
sig_PrioritylevelSet
sig_PrioritylevelGet
sig_Catch
sig_Release
sig_Broadcast
sig_Send
sig_Ignore
sig_Block
sig_Unblock

Prior GUI Framework

Prior Shell

File(s): Prior_shell.c

The Prior Shell provides a list of shell commands that allow for debugging and direct control over the kernel. Shell access is password protected, this password can be set in the configuration file; CFG_SHELLPW.

shutdown -<mode>

Shuts the operating system down

Argument	*Description*
-h	*halt after shutdown*
-r	*reboot after shutdown*

runtime

Displays the current runtime in HH:SS

Argument	*Description*
N/A	-

osfreq ("x")

Displays or sets the OS frequency

Argument	*Description*
"x"	*Sets the OS frequency to x Hz*
	Gets the OS frequency in Hz

schedulerlock -<mode> *Locks or unlocks the scheduler*

Argument	*Description*
-l	*Locks the scheduler*
-u	*Unlocks the scheduler*

displist -<list> -<format> *Displays the selected task list in the selected format*

Argument	*Description*
-t	*Select TCB list to display*
-w	*Select TCB Wait list to display*
-e	*Select Execution Queue to display*

Argument	*Description*
-r	*Select raw format (TaskTrace)*
-c	*Select command line format*

memnfo *Displays information about the OS memory*

Argument	*Description*
N/A	-

poolnfo "x" *Displays information about pool x*

Argument	*Description*
"x"	*Pool number; 0 ≤ x ≤ CFG_N_POOLS*

run –<identifier type> "identifier" *Signals the scheduler to run the given task*

Argument	*Description*
-id "ID"	*Task ID*
-gn"taskname"	*Generic task name*

TaskTrace

TaskTrace is a debugging tool for Prior RTOS allowing the developer to monitor and control the kernel. The target provides the software back-end running on a PC with detailed information on the kernel objects that are being watched.

In Depth: Schedulers

Cooperative

Preemptive

Implementation example