

**Prior RTOS V 0.3.X**

User Manual

\*not complete\*

Copyright© 2016 D. van de Spijker

# Table of Contents

[Table of Contents 2](#_Toc474272895)

[List of Figures 6](#_Toc474272896)

[License and distribution 7](#_Toc474272897)

[Abbreviations and definitions 8](#_Toc474272898)

[Introduction 9](#_Toc474272899)

[What is Prior? 9](#_Toc474272900)

[Core principles 9](#_Toc474272901)

[Customizability 9](#_Toc474272902)

[Priority 9](#_Toc474272903)

[Intuitiveness 9](#_Toc474272904)

[Specifications and Features 10](#_Toc474272905)

[High Level Architecture 13](#_Toc474272906)

[OS definitions 14](#_Toc474272907)

[Types 14](#_Toc474272908)

[Macros 15](#_Toc474272909)

[Prior Configuration file 16](#_Toc474272910)

[Global settings 16](#_Toc474272911)

[Memory Management settings 17](#_Toc474272912)

[Module settings 17](#_Toc474272913)

[Additional settings 18](#_Toc474272914)

[Peripheral settings 18](#_Toc474272915)

[Architecture Port 19](#_Toc474272916)

[port\_OSTimerInit 19](#_Toc474272917)

[port\_OSTimerStart 19](#_Toc474272918)

[port\_OSTimerStop 19](#_Toc474272919)

[port\_OSTimerReset 19](#_Toc474272920)

[port\_OSTimerCountGet 19](#_Toc474272921)

[port\_OSTickStart 19](#_Toc474272922)

[port\_OSTickStop 19](#_Toc474272923)

[port\_GlobalInterruptsDisable 19](#_Toc474272924)

[port\_GlobalInterruptsEnable 19](#_Toc474272925)

[port\_OSInterruptDisable 19](#_Toc474272926)

[port\_OSInterruptEnable 19](#_Toc474272927)

[port\_WatchdogDisable 19](#_Toc474272928)

[port\_WatchdogEnable 19](#_Toc474272929)

[Deployment 20](#_Toc474272930)

[Task Priority System 21](#_Toc474272931)

[OS Objects and the API 22](#_Toc474272932)

[Kernel Control 22](#_Toc474272933)

[os\_Init 22](#_Toc474272934)

[os\_Start 23](#_Toc474272935)

[os\_FrequencySet 24](#_Toc474272936)

[os\_FrequencyGet 24](#_Toc474272937)

[os\_VersionGet 25](#_Toc474272938)

[os\_RuntimeGet 25](#_Toc474272939)

[os\_TickTGet 26](#_Toc474272940)

[os\_CritSectEnter 27](#_Toc474272941)

[os**\_**CritSectExit 28](#_Toc474272942)

[os\_ISREnter 29](#_Toc474272943)

[os\_ISRExit 29](#_Toc474272944)

[os\_SchedulerLock 30](#_Toc474272945)

[os\_SchedulerUnlock 30](#_Toc474272946)

[Memory Management 31](#_Toc474272947)

[mm\_OSHeapFreespaceGet 31](#_Toc474272948)

[mm\_PoolCreate 32](#_Toc474272949)

[mm\_PoolDelete 33](#_Toc474272950)

[mm\_PoolFormat 34](#_Toc474272951)

[mm\_PoolDefrag 35](#_Toc474272952)

[mm\_PoolMove 35](#_Toc474272953)

[mm\_PoolSort 35](#_Toc474272954)

[mm\_PoolHealthGet 36](#_Toc474272955)

[mm\_PoolFreespaceGet 36](#_Toc474272956)

[mm\_CollectorEnable 37](#_Toc474272957)

[mm\_CollectorDisable 37](#_Toc474272958)

[mm\_StcAlloc 37](#_Toc474272959)

[mm\_DynAlloc 38](#_Toc474272960)

[mm\_ReAlloc 39](#_Toc474272961)

[mm\_Free 40](#_Toc474272962)

[Tasks 41](#_Toc474272963)

[task\_Create 44](#_Toc474272964)

[task\_Delete 45](#_Toc474272965)

[task\_Wait 46](#_Toc474272966)

[task\_EventHandle 47](#_Toc474272967)

[task\_Wake 48](#_Toc474272968)

[task\_Suspend 49](#_Toc474272969)

[task\_SuspendAll 50](#_Toc474272970)

[task\_Resume 51](#_Toc474272971)

[task\_ResumeAll 51](#_Toc474272972)

[task\_Sleep 51](#_Toc474272973)

[task\_Delay 51](#_Toc474272974)

[task\_Yield 51](#_Toc474272975)

[task\_GenericNameSet 51](#_Toc474272976)

[task\_GenericNameGet 51](#_Toc474272977)

[task\_ArgumentSet 51](#_Toc474272978)

[task\_PrioritySet 52](#_Toc474272979)

[task\_PriorityGet 53](#_Toc474272980)

[task\_CategorySet 53](#_Toc474272981)

[task\_CategoryGet 53](#_Toc474272982)

[task\_StateGet 53](#_Toc474272983)

[task\_RuntimeGet 53](#_Toc474272984)

[task\_RuntimeReset 53](#_Toc474272985)

[Timers 54](#_Toc474272986)

[timer\_Create 56](#_Toc474272987)

[timer\_Delete 57](#_Toc474272988)

[timer\_Start 57](#_Toc474272989)

[timer\_Stop 57](#_Toc474272990)

[timer\_Pause 57](#_Toc474272991)

[timer\_Reset 57](#_Toc474272992)

[timer\_StartAll 57](#_Toc474272993)

[timer\_StopAll 57](#_Toc474272994)

[timer\_ResetAll 57](#_Toc474272995)

[timer\_StateGet 57](#_Toc474272996)

[timer\_TicksGet 57](#_Toc474272997)

[timer\_IntervalSet 57](#_Toc474272998)

[timer\_IntervalGet 57](#_Toc474272999)

[timer\_IterationsSet 58](#_Toc474273000)

[timer\_IterationsGet 58](#_Toc474273001)

[timer\_ParameterGet 58](#_Toc474273002)

[timer\_ParameterSet 58](#_Toc474273003)

[Event Groups 59](#_Toc474273004)

[eventgrp\_Create 59](#_Toc474273005)

[eventgrp\_Delete 59](#_Toc474273006)

[eventgrp\_FlagsSet 59](#_Toc474273007)

[eventgrp\_FlagsClear 59](#_Toc474273008)

[eventgrp\_FlagsGet 59](#_Toc474273009)

[eventgrp\_Broadcast 59](#_Toc474273010)

[Pipes 60](#_Toc474273011)

[pipe\_Open 60](#_Toc474273012)

[pipe\_Close 60](#_Toc474273013)

[pipe\_Write 60](#_Toc474273014)

[pipe\_Read 60](#_Toc474273015)

[pipe\_DataLengthGet 60](#_Toc474273016)

[pipe\_Flush 60](#_Toc474273017)

[Semaphores & Mutexes 61](#_Toc474273018)

[Ring-buffers 62](#_Toc474273019)

[Mailboxes 64](#_Toc474273020)

[Asynchronous Signals 65](#_Toc474273021)

[Prior GUI Framework 66](#_Toc474273022)

[Prior Shell 67](#_Toc474273023)

[TaskTrace 69](#_Toc474273024)

[In Depth: Schedulers 69](#_Toc474273025)

[Cooperative 69](#_Toc474273026)

[Preemptive 69](#_Toc474273027)

[Implementation example 69](#_Toc474273028)

# List of Figures

# License and distribution

Prior RTOS is distributed under the MIT License.

The MIT License (MIT)

Copyright© 2016 D. van de Spijker

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

1. The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.
2. The name of Prior RTOS may not be used to endorse or promote products derived from this Software without specific prior written permission.
3. This Software may only be redistributed and used in connection with a product in which Prior RTOS is integrated. Prior RTOS shall not be distributed, under a different name or otherwise, as a standalone product.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

# 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 |  |  |

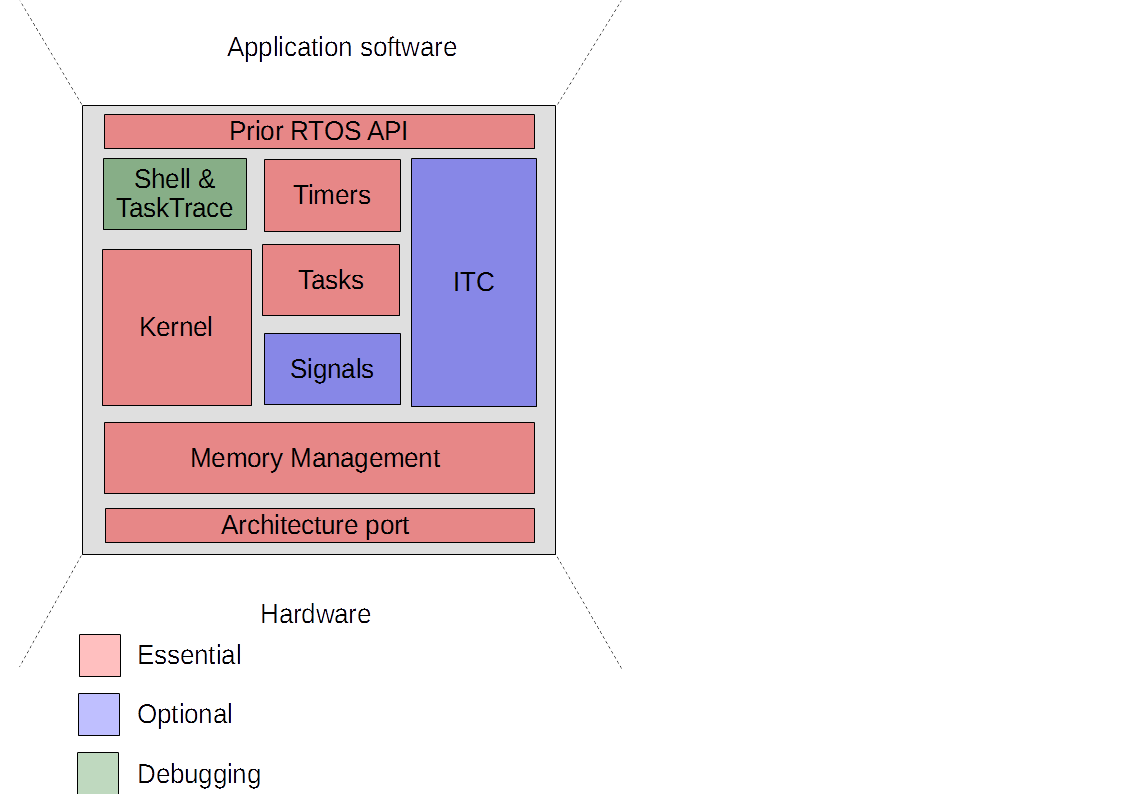
* Data memory scalability per unit:

|  |  |
| --- | --- |
| 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 files of the selected port (PriorRTOS\ports\<manufacturer>\<target>\Port.c/.h) to \include\hal\port.

**Step 1:** 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 2:** Include the Prior RTOS main header in the file where the OS is used. (PriorRTOS\Include\Prior\_RTOS.h) .

**(Step 3 :)** 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)* handlerTask 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)* frequencyNew 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 byteslocated in the OS heap. After creating a pool, allocations can be made inside this pool.

**Parameters:***(U32\_T)* pool\_size Size of the new poolin 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

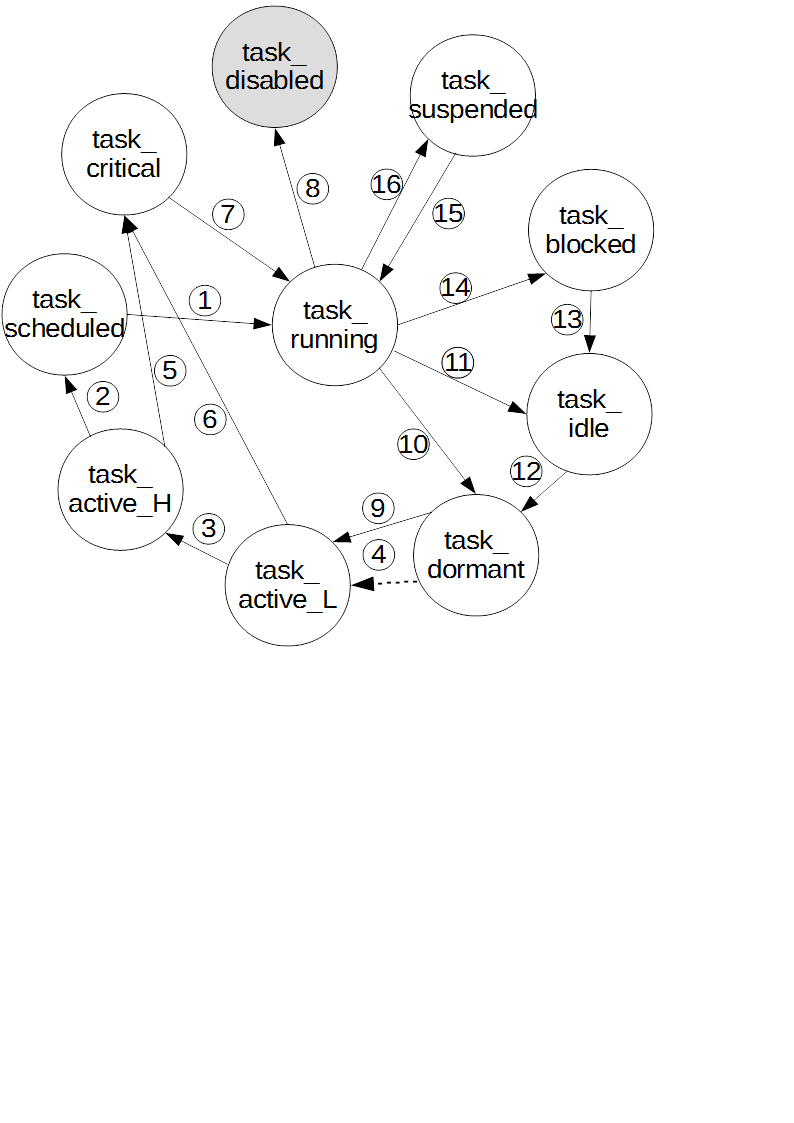
Tasks are one of the essential objects within Prior. They enable the programmer to split their code into smaller parts. Within these smaller parts code is sequential as usual, but the smaller parts are given execution time by the OS kernel. Of course there are ways to influence the task-flow like synchronization, 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)* handlerAddress 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)*handlerAddress 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 )* handlerTask entry point; NULL to address the currently running task

*(ID\_t)* object\_IDWait 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)* handlerAddress 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)* handlerAddress 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)* handlerTask handler start address

*(Prio\_t)* priorityNew 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)* handlerTask 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.

  
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**

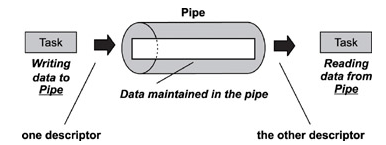
### **eventgrp\_FlagsClear**

### **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

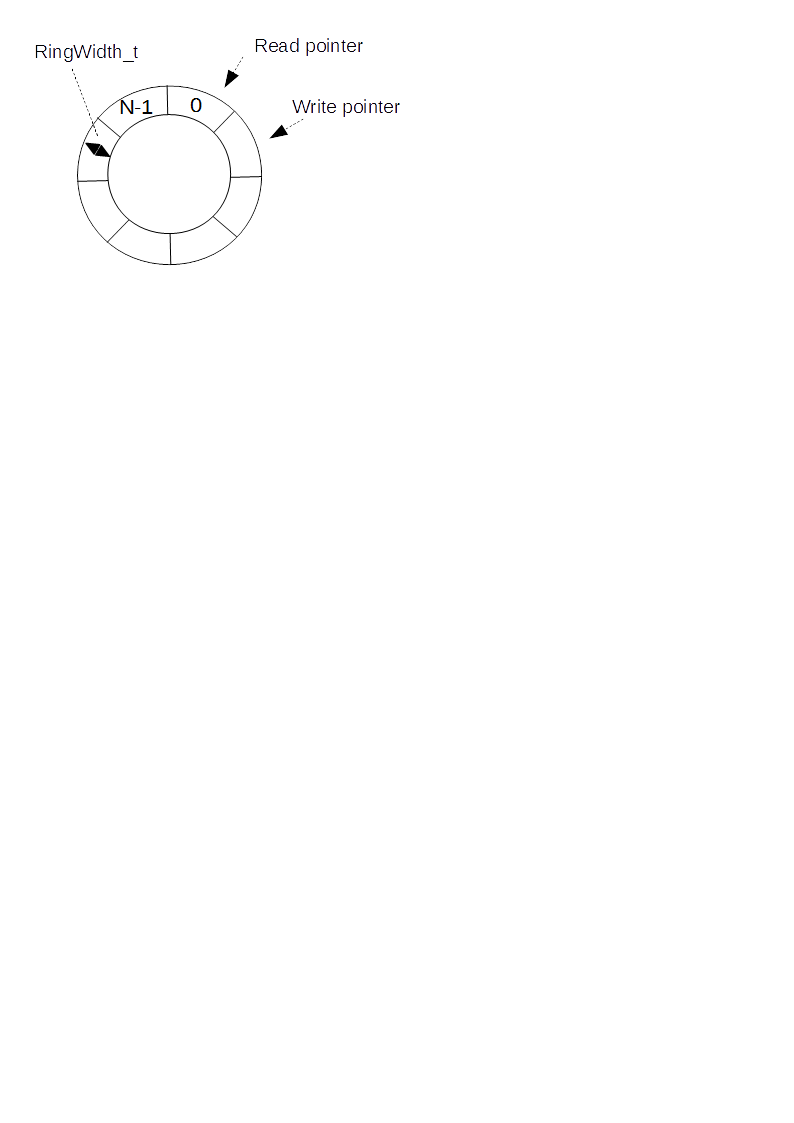
Semaphores and Mutexes are small kernel objects used to lock entities and other kernel objects. These locks can be used to keep tasks from accessing a (shared) resource.   
Prior distinguishes 4 kinds:   
- Binary semaphore; not owned by specific task, single flag **(type 1)**  
- Counting semaphore; 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.

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

**shutdown -<mode>** *Shuts the operating system down*

|  |  |
| --- | --- |
| Argument | Description |
| N/A | *-* |

**runtime** *Displays the current runtime in HH:SS*

|  |  |
| --- | --- |
| Argument | Description |
| **“x”** | *Sets the OS frequency to x Hz* |
|  | *Gets the OS frequency in Hz* |

**osfreq (“x”)** *Displays or sets the OS frequency*

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

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

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

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

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

**memnfo** *Displays information about the OS memory*

|  |  |
| --- | --- |
| Argument | Description |
| N/A | - |

|  |  |
| --- | --- |
| Argument | Description |
| **“x”** | *Pool number; 0 ≤ x ≤ CFG\_N\_POOLS* |

**poolnfo “x”** *Displays information about pool x*

|  |  |
| --- | --- |
| Argument | Description |
| **-id “ID”** | *Task ID* |
| **-gn”taskname”** | *Generic task name* |

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

# 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