TNKernel

Real-Time Kernel

v. **2**

(http://www.tnkernel.com/)

Document Disclaimer

The information in this document is subject to change without notice. While the information herein is assumed to be accurate, Yuri Tiomkin (the author) assumes no responsibility for any errors or omissions.
The author makes and you receive no warranties or conditions, express, implied, statutory or in any communications with you. The author specifically disclaims any implied warranty of merchantability or fitness for a particular purpose.

Copyright notice

Trademarks

Document Version:

- 2.3

Acknowledgments:

Thanks to Clemens Fischer – for reading and providing corrections

INTRODUCTION

TNKernel is a compact and very fast real-time kernel for the embedded 32/16 bits microprocessors. **TNKernel** performs a preemptive priority-based scheduling and a round-robin scheduling for the tasks with identical priority.
The current version of **TNKernel** includes semaphores, mutexes, data queues, event flags and fixed-sized memory pools. The system functions calls in the interrupts are supported.
TNKernel is a fully portable (written mostly in ANSI C except processor-specific parts), but the current version of **TNKernel** has been ported for the ARM microprocessors only.

TNKernel has been written “under the significant influence” of the µITRON 4.0 Specifications.

The µITRON 4.0 Specifications is an open real-time kernel specification developed by the ITRON Committee of the TRON Association.The µITRON 4.0 Specification document can be obtained from the ITRON Project web site (http://www.assoc.tron.org/eng/document.html).

TNKernel is distributed in the source code form free of charge under the FreeBSD-like license.

TASKS

In **TNKernel**, a task is a branch of the code that runs concurrently with another tasks from the programmer's point of view. At the physical level, tasks are actually executed using processor time sharing. Each task can be considered to be an independed program, which executes in its own context (processor registers, stack pointer, etc.).
When the currently running task loses its claim for executing (by the issuing of a system call or interrupt), a context switch is performed. The current context (processor registers, stack pointer, etc.) is saved and the context of another task is restored. This mechanism in the **TNKernel** is called the "dispatcher".
Generally, there are more than one executable task, and it is necessary to determine the order of the task switching (execution) by using some rules. "Scheduler" is a mechanism that controls the order of the task execution.
TNKernel uses a priority-based scheduling based on a priority level assigned to the each task. The smaller the value of the priority, the higher the priority level. **TNKernel** uses a 32 levels of priority.
Priorities 0 (highest) and 31 (lowest) are reserved by the system for the internal using. The user may create tasks with priorities 1…30.
In **TNKernel**, more than one task can have the same (identical) priority.

TASK STATES

There are four task states in **TNKernel**:

1. RUNNING state

    The task is currently executing.

2. READY state

    The task is ready to execute, but cannot do so because a task with higher priority (sometimes same priority) is already executing. A task may execute at any time once the processor becomes available.
    In **TNKernel**, both RUNNING and READY states are marked as RUNNABLE.

3. WAIT/SUSPEND state

    When a task is in the WAIT/SUSPEND state, the task cannot execute because the conditions necessary for its execution have not yet been met and the task is waiting for them. When a task enters the WAIT/SUSPEND state, the task's context is saved. When the task resumes execution from the WAIT/SUSPEND state, the task's context is restored.

WAIT/SUSPEND actually have one of three types:

- WAITING state

The task execution is blocked until some synchronization action occurs, such as timeout expiration, semaphore available, event occurring, etc.

- SUSPENDED state

The task is forced to be blocked (switched to the non-executing state) by another task or itself.

- WAITING_SUSPENDED state

Both WAITING and SUSPENDED states co-exist.
In **TNKernel**, if a task leaves a WAITING state, but a SUSPENDED state exists, the task is not switched to the READY/RUNNING state. Similarly, if a task leaves SUSPENDED state, but a WAITING state exists, the task is not switched to the READY/RUNNING state. A task is switched to READY/RUNNING state only if there are neither WAITING nor SUSPENDED states flagged on it.

4. DORMANT state

The task has been initialized and it is not yet executing or it has already exited. Newly created tasks always begin in this state.

SCHEDULING RULES

In **TNKernel**, as long as the highest privilege task is running, no other task will execute unless the highest privilege task cannot execute (for instance, for being placed in the WAITING state).
Among tasks with different priorities, the task with the highest priority is the highest privilege task and will execute.
Among tasks of the same priority, the task that entered into the runnable (RUNNING or READY) state first is the highest privilege task and will execute.

Example: Task A has priority 1, tasks B, C, D, E have priority 3, tasks F,G have priority 4, task I has priority 5.
If all tasks are in the READY state, this is the sequence of tasks executing :

1. Task A - highest priority (priority 1)
2. Tasks B, C, D, E - in order of entering into runnable state for this priority (priority 3)
3. Tasks F, G - in order of entering into runnable state for this priority (priority 4)
4. Task I - lowest priority (priority 5)

In **TNKernel**, tasks with the same priority may be scheduled in round robin fashion by getting a predetermined time slice for each task with this priority.

INTERRUPTS

In **TNKernel**, there are special functions for processing system calls inside interrupt(s).Generally, if some conditions, checked while in interrupt, required context switching, system does it according to the architecture of processor (some processors use different stack to service interrupts).

SYSTEM TASKS

In **TNKernel**, the task with priority 0 (highest) is used for supporting the system tick timer functionality and the task with priority 31 (lowest) is used for performing statistics.
TNKernel automatically creates these tasks at the system start.

TNKernel FUNCTIONALITY

1.Tasks

The user may create tasks with priorities 1…30. User tasks should newer communicate with tasks of priorities 0 and 31 (for instance, to attempt to switch these tasks into suspend state etc.).
The system will reject any attempt to create a task with priority 0 or 31.
More than one user tasks can have the same priority. Tasks with identical priorities have the ability for round-robin scheduling.

Task functions (**TNKernel** version 2.x)

Function	Description
tn_task_create	Create task
tn_task_terminate	Move task to DORMANT state
tn_task_exit	Terminate currently running task
tn_task_delete	Delete already terminated task
tn_task_activate	Activate task. Task is switched from DORMANT state to runnable state
tn_task_iactivate	The same as above, but in interrupts
tn_task_change_priority	Change current task priority
tn_task_suspend	Suspend task. If task is runnable, it is switched to the SUSPENDED state. If task is in the WAITING stage, it is moved into the WAITING_SUSPENDED state
tn_task_resume	Resume suspended task - allows the task to continue its normal processing.
tn_task_sleep	Move currently running task sleep.
tn_task_wakeup	Wake up the task from sleep.
tn_task_iwakeup	The same as above, but in interrupts.
tn_task_release_wait	Forcibly release task from waiting (including sleep), but not from the SUSPENDED state
tn_task_irelease_wait	The same as above, but in interrupts

2. Semaphores

A semaphore has a resource counter and a wait queue. The resource counter shows the number of unused resources. The wait queue manages the tasks waiting for resources from this semaphore. The resource counter is incremented by 1 when a task releases a semaphore resource, and is decremented by 1 when a task acquires a semaphore resource.
If a semaphore has no available resources (resource counter is 0), a task that requested a resource will wait in the semaphore wait queue until a resource is arriving (another task releases it to the semaphore).

Semaphore functions (**TNKernel** version 2.x)

Function	Description
tn_sem_create	Create semaphore
tn_sem_delete	Delete semaphore
tn_sem_signal	Release semaphore resource
tn_sem_isignal	The same as above, but in interrupts
tn_sem_acquire	Acquire one resource from semaphore
tn_sem_polling	Acquire one resource from semaphore with polling
tn_sem_ipolling	The same as above, but in interrupts

3. Mutexes

A mutex is an object used for mutual exclusion of a shared resource.
Mutex supports two approaches for avoiding the unbounded priority inversions problem - the priority inheritance protocol and the priority ceiling protocol. A discussion about strengths and weaknesses of each protocol as well as priority inversions problem is beyond the scope of this document.

A mutex has a similar functionality as a semaphore with maximum count = 1(a binary semaphore). The differences are that a mutex can only be unlocked by the task that locked it and that a mutex is unlocked by **TNKernel** when the locking task terminates.

A mutex uses the priority inheritance protocol when it has been created with the TN_MUTEX_ATTR_INHERIT attribute, and the priority ceiling protocol when its attribute value is TN_MUTEX_ATTR_CEILING.

The mutexes in **TNKernel** support full-feature priority inheritance protocols according to the document [1]. There is a difference in approach to the µITRON 4.0 Specification: µITRON 4.0 proposes a subset of the priority ceiling protocol (a highest locker protocol), **TNKernel** uses a full version of priority ceiling protocol.

The priority inheritance protocol solves the priority inversions problem but doesn't prevents deadlocks.

The priority ceiling protocol prevents deadlocks and chained blocking but it is slower than the priority inheritance protocol .

Mutex functions (**TNKernel** version 2.x)

Function	Description
tn_mutex_create	Create a mutex
tn_mutex_delete	Delete a mutex
tn_mutex_lock	Lock a mutex
tn_mutex_lock_polling	Try to lock a mutex (with polling)
tn_mutex_unlock	Unlock a mutex

4. Data Queues

A data queue is a FIFO that stores pointer (of type *void**) in each cell, called (in µITRON style) a data element. A data queue also has an associated wait queue each for sending (wait_send queue) and for receiving (wait_receive queue).

A task that sends a data element is tried to put the data element into the FIFO. If there is no space left in the FIFO, the task is switched to the WAITING state and placed in the data queue's wait_send queue until space appears (another task gets a data element from the data queue).

A task that receives a data element tries to get a data element from the FIFO. If the FIFO is empty (there is no data in the data queue), the task is switched to the WAITING state and placed in the data queue's wait_receive queue until data element arrive (another task puts some data element into the data queue).

To use a data queue just for the synchronous message passing, set size of the FIFO to 0.

The data element to be sent and received can be interpreted as a pointer or an integer and may have value 0 (NULL).

Data Queue functions (**TNKernel** version 2.x)

Function	Description
tn_queue_create	Create data queue
tn_queue_delete	Delete data queue
tn_queue_send	Send (put) a data element into the data queue
tn_queue_send_polling	Try to send(put) a data element into the data queue(with polling)
tn_queue_isend_polling	The same as above, but inside interrupts
tn_queue_receive	Receive (get) a data element from the data queue
tn_queue_receive_polling	Try to receive (get) a data element from the data queue (with polling)
tn_queue_ireceive	The same as above, but inside interrupts

5. Eventflags

An eventflag has an internal variable (of size integer), which is interpreted as a bit pattern where each bit represents an event. An eventflag also has a wait queue for the tasks waiting on these events.

A task may set specified bits when an event occurs and may clear specified bits when necessary. A task waiting for events to occur will wait until every specified bit in the eventflag bit pattern is set. The tasks waiting for an eventflag are placed in the eventflags wait queue.
An eventflag is a very suitable synchronization object for cases where (for some reasons) one task has to wait for many tasks, or vice versa, many tasks have to wait for one task.

Eventflag functions (**TNKernel** version 2.x)

Function	Description
tn_event_create	Create eventflag
tn_event_delete	Delete eventflag
tn_event_wait	Wait until eventflag satisfies the release condition
tn_event_wait_polling	Wait until eventflag satisfies the release condition, with polling
tn_event_iwait	The same as above, but inside interrupts
tn_event_set	Set eventflag
tn_event_iset	The same as above, but inside interrupts
tn_event_clear	Clears the bits in the eventflag
tn_event_iclear	The same as above, but inside interrupts

6. Fixed-Sized Memory Pools

A fixed-sized memory pool is used for managing fixed-sized memory blocks dynamically. A fixed-sized memory pool has a memory area where fixed-sized memory blocks are allocated and the wait queue for acquiring a memory block.
If there are no free memory blocks, a task trying to acquire a memory block will be placed into the wait queue until a free memory block arrives (another task returns it to the memory pool).

Fixed-sized memory pool functions (**TNKernel** version 2.x)

Function	Description
tn_fmem_create	Create Fixed-Sized Memory Pool
tn_fmem_delete	Delete Fixed-Sized Memory Pool
tn_fmem_get	Acquire (get) a memory block from pool
tn_fmem_get_polling	Acquire (get) a memory block from pool, with polling
tn_fmem_get_ipolling	The same as above, but inside interrupts
tn_fmem_release	Release (put back to pool) a memory block
tn_fmem_irelease	The same as above, but inside interrupts

STARTING TNKernel

For the **TNKernel**, the *main()* function will look like:

```
int main()
{
   //-- Operations before TNKernel's start (for instance, hardware
                                                     initialization)
         .
         .
         .
   tn_start_system();    //-- Never returns
   return 0;             //-- Unreachable
}
```

The **tn_start_system()** function performs all actions to initialize and start **TNKernel** (initializes the system global variables, creates tasks with priorities 0 and 31, calls start-up functions etc.)
The **tn_start_system()** function internally calls the **tn_app_init()** function**.**
The contents (body) of the **tn_app_init()** function has to be defined by the user and may be empty. In this function the user has to create all tasks, semaphores, data queues, memory pools etc., which he wants to have before the system start.

TNKernel TIME TICKS

For the purpose of calculating timeouts and delays, **TNKernel** uses a time tick timer. In **TNKernel**, this time tick timer must to be a some kind of hardware timer that produces interrupt for time ticks processing. The period of this timer is user determined (usually in the range 0.5...20 ms).
Within the time ticks interrupt processing, it is only necessary to call the **tn_tick_int_processing()** functions (see details below).
To minimize interrupt processing time, **TNKernel** makes the most time consuming processing inside the task with priority 0. The **tn_tick_int_processing()** function releases the priority 0 task from sleep (see file tn.c).

ROUND ROBIN SCHEDULING IN TNKernel

TNKernel has the ability to make round robin scheduling for tasks with identical priority.
By default, round robin scheduling is turned off for all priorities. To enable round robin scheduling for tasks on certain priority level and to set time slices for these priority, user must call the **tn_sys_tslice_ticks()** function.
The time slice value is the same for all tasks with identical priority but may be different for each priority level.
If the round robin scheduling is enabled, every system time tick interrupt increments the currently running task time slice counter.
When the time slice interval is completed, the task is placed at the tail of the ready to run queue of its priority level (this queue contains tasks in the RUNNABLE state) and the time slice counter is cleared. Then the task may be preempted by tasks of higher or equal priority.

If tasks with round-robin scheduling never switch to the WAITING state (for instance when there are no semaphore(s) acquiring, sleep,etc.), lower priority tasks will never run !

In most cases, there is no reason to enable round robin scheduling. For applications running multiple copies of the same code, however, (GUI windows, etc.), round robin scheduling is an acceptable solution.

TNKernel PORT

There are few files in **TNKernel** having processor-depended contents:

tn_port.h

This file includes definitions and macros for the processor's memory alignment.

tn_port.c

This file contains functions:

tn_stack_init	Creates task stack frame. System invokes it at task creation time.
get_task_by_timer_queque	Calculates task TCB start address by address of task's timer queue
get_task_by_tsk_queue	Calculates task TCB start address by address of task's wait queue
get_task_by_block_queque	Calculates task TCB start address by address of entry in the system blocked task list - uses only for the mutexes priority seiling protocol
get_mutex_by_mutex_queque	Calculates mutex TN_MUTEX start address by the mutex entry in the task's locked mutexes list
get_mutex_by_lock_mutex_queque	Calculates mutex TN_MUTEX start address by the mutex entry in the system's locked mutexes list
get_mutex_by_wait_queque	Calculates mutex TN_MUTEX start address by the list of the tasks that wait on a mutex

tn_arm_port.s

For different compilers, this file has different names: tn_port_asm_armcc.s, tn_port_asm_ghs.s, tn_port_asm_keil.s, tn_port_asm.s79, etc.

This file contains assembly-written functions:

Function	C-language prototype
tn_switch_context	void tn_switch_context (void)
tn_switch_context_exit	void tn_switch_context_exit(void)
tn_cpu_irq_isr	void tn_cpu_irq_isr (void)
tn_cpu_save_sr	int tn_cpu_save_sr (void)
tn_cpu_restore_sr	void tn_cpu_restore_sr (int *sr*)
tn_start_exe	void tn_start_exe (void)
tn_chk_irq_disabled	int tn_chk_irq_disabled (void)
ffs_asm	int ffs_asm (unsigned int *val*)
tn_cpu_fiq_isr	void tn_cpu_fiq_isr (void)

All assembly-written functions are for the system internal purposes only. There is no reason to invoke them from the user tasks.

The **tn_switch_context()** function performs system context switch outside of interrupts.

The **tn_switch_context_exit()** function is used to exit from currently running task.

The **tn_cpu_irq_isr()** function is invoked by a hardware to process ARM IRQ interrupt. It calls the

tn_cpu_irq_handler() function to execute user code for interrupt processing. Then the system checks context switch condition and - if necessary - performs a context switch within an interrupt.

The **tn_cpu_fiq_isr()** function is invoked by hardware to process ARM FIQ interrupt similar to the function **tn_cpu_irq_isr()**.

The **tn_cpu_save_sr()** function saves the contents of the ARM SPSR processor register in the additional variable.

The **tn_cpu_restore_sr()** function restores contents of ARM SPSR register previously saved by function **tn_cpu_save_sr().**

The **tn_start_exe()** function makes first context switching at the system start.

The **tn_chk_irq_disabled()** function checks the ARM CPSR processor register for the current IRQ status and returns 1 if IRQ interrupt is disabled, otherwise returns 0.

The **ffs_asm()** function implements "find first set (bit)" operation (starting from LSB). It returns bit position (1…32) if bit with value '1' is found, otherwise returns 0.

startup_hardware_init.s

This file contains assembly-written functions:

Function	**C-language prototype**
tn_startup_hardware_init	Not used in C
tn_arm_disable_interrupts	void tn_arm_disable_interrupts (void)
tn_arm_enable_interrupts	void tn_arm_enable_interrupts (void)

The **tn_startup_hardware_init** routine is called immediately after reset to setup hardware that is vital for the processor's functionality (for instance, SDRAM controller setup, PLL setup, etc.)
It is assumed that other hardware initialization routine(s) will be invoked later by the C-language function calls.

The **tn_arm_enable_interrupts()** function enables interrupts in ARM processors. It replaces a similar function with different names in different compilers (for instance, in IAR ARM compiler: __enable_interrupt(), in Rowley CrossWorks: __ARMLIB_enableIRQ(), etc.)

The ***tn_arm_disable_interrupts ()*** function complements the **tn_arm_enable_interrupts()** function.

tn_user.c

This file includes functions:
- *tn_cpu_irq_handler*
- *tn_cpu_int_enable*

The **tn_cpu_irq_handler()** function is the user routine to process interrupts. The user has to include custom code for the handling interrupts. Example code contained in the distribution may be used as a guide.
TNKernel invokes this function internally from the **tn_cpu_irq_isr()** function or the **tn_cpu_fiq_isr()** function without user intervention (see above).

The sample code for the ARM processor family uses polling to recognize an interrupt sources. An alternative solution would be to read registers in the interrupt controller. A discussion about strengths and weaknesses of these approaches is beyond the scope of this document.

For instance, the **tn_cpu_irq_handler()** function may be like this:

```
void tn_cpu_irq_handler(void)
{
   if(...)      //-- Int source - time ticks timer
   {
      tn_tick_int_processing();  //-- Mandatory!
      .
      .
      .
   }
   else if(...) //-- Int source - UART
   {
      .
      .
      .
   }

   else if(...) //-- Int source - SPI
   {
      .
      .
      .
   }
   //-- etc.
}
```

or like this (example for the STMicroelectronics STR71X ARM):

```
void tn_cpu_irq_handler(void)
{
   volatile int ivr;

   ivr = rEIC_IVR;              //-- For correct interrupts controller
                                //-- hardware functionality only
   ivr = rEIC_CICR;             //-- Get interrupt number
   if(ivr == IVECT_TIMER1)
      timer1_irq_func();        //-- The user function to handle Timer 1 int
   else if(ivr == IVECT_UART0)
      uart0_irq_func();         //-- The user function to handle UART0 1 int
       .
       .
       .
}
```

It is important that the **tn_tick_int_processing()** function has to be invoked for the system time ticks timer interrupt processing within the **tn_cpu_irq_handler()** function or within timer interrupt handling function.

The **tn_cpu_int_enable()** function enables all interrupts for vectors utilized by the user project and than enables global interrupts. The user must enable the system time ticks interrupt in this function. **TNKernel** calls this function without user intervention.

For instance, the **tn_cpu_int_enable()** function may be like this:

```
void tn_cpu_int_enable()
{
   //-- Enable UART interrupt
         .
         .
         .
```

```
    //-- Enabled timer interrupt for time ticks (mandatory).
           .
           .
           .
    //-- Enable DMA interrupt
           .
           .
           .
    //-- Enable global interrupts
           .
           .
           .
}
```

TNKernel API FUNCTIONS

System functions

tn_sys_tslice_ticks	Enable/disable round robin scheduling

Function:

int tn_sys_tslice_ticks (int *priority*,
int *value***)**

Parameter:

priority	Priority for which round-robin scheduling is enabled/disabled
value	Time slice value. Must be greater than 0 and less or equal to the MAX_TIME_SLICE. If *value* is NO_TIME_SLICE, round-robin scheduling for tasks with *priority* is disabled.

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Input parameter(s) has a wrong value

Description:

This function controls round-robin scheduling for the tasks with a given *priority.*
A time slice *value* is calculated in the system ticks. The *value* is the same for all tasks with identical *priority* but may be different for each priority level.

Tasks functions

Each task has an associated task control block (TCB), defined (in file tn.h) as:

```
typedef struct _TN_TCB        //-- Ver. 2.x
{
   unsigned int * task_stk;    //-- Pointer to the task's top of stack
   CDLL_QUEUE task_queue;      //-- Queue to include task in the ready/wait lists
   CDLL_QUEUE timer_queue;     //-- Queue to include task in the timer(timeout,etc.) list
   CDLL_QUEUE block_queue;     //-- Queue to include task in the blocked task list only
                               // used for mutexes priority seiling protocol
   CDLL_QUEUE create_queue;    //-- Queue is used to include task in create list only
   CDLL_QUEUE mutex_queue;     //-- List of all mutexes locked by the tack (ver 2.x)
   CDLL_QUEUE * pwait_queue;   //-- Ptr to the object's (semaphor,event,etc.) wait list,
                               // the task is waiting for (ver 2.x)
   struct _TN_TCB * blk_task;  //-- Store task  blocking our task (for the mutexes
                               // priority ceiling protocol only (ver 2.x)

   int * stk_start;            //-- Base address of the task's stack space
   int   stk_size;             //-- The task stack size (in sizeof (void*), not bytes)
   void * task_func_addr;      //-- filled on creation  (ver 2.x)
   void * task_func_param;     //-- filled on creation  (ver 2.x)

   int  base_priority;         //-- Task base priority  (ver 2.x)
   int  priority;              //-- Task current priority
   int  id_task;               //-- ID for verification(is it a task or another object?)
                               // All tasks have the same id_task magic number (ver 2.x)

   int  task_state;            //-- Task state
   int  task_wait_reason;      //-- Reason for the waiting
   int  task_wait_rc;          //-- Waiting return code (reason why waiting finished)
   int  tick_count;            //-- Remaining time until timeout
   int  tslice_count;          //-- Time slice counter

   int  ewait_pattern;         //-- Event wait pattern
   int  ewait_mode;            //-- Event wait mode:  _AND or _OR

   void * data_elem;           //-- Location to store data queue entry, if the data queue is full

   int  activate_count;        //-- Activation request count - for statistic
   int  wakeup_count;          //-- Wakeup request count - for statistic
   int  suspend_count;         //-- Suspension count - for statistic

// Other implementation specific fields may be added below

}TN_TCB;
```

tn_task_create Create Task

Function:

int tn_task_create (TN_TCB * *task*,
void (*task_func***) (void ***param***)**,
int *priority***,**
unsigned int * *task_stack_start***,**
int *task_stack_size***,**
void * *param***,**
int *option* **)**

Parameters:

task	Pointer to the task TCB to be created
task_func	Task body function. This is the address of a function declared as: **void task_func (void * param)**
priority	Task priority. User tasks may have priorities 1…30 (system uses priorities 0 and 31 for internal purposes)
task_stack_start	Task's stack bottom address
task_stack_size	Task's stack size – number of task stack elements (not bytes)
param	*task_func* parameter. *param* will be passed to *task_func* on creation time
option	Creation option. Option values: 0 After creation task has a DORMANT state (needs *tn_task_activate*() call for activation) TN_TASK_START_ON_CREATION After creation task is switched to the runnable state (READY/RUNNING)

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Input parameter(s) has a wrong value

Description:

This function creates a *task*. A field *id_task* of the structure *task* must be set to 0 before invoking this function. A memory for the *task* TCB and a *task* stack must be allocated before the function call. An allocation may be static (global variables of the `TN_TCB` type for the *task* and `unsigned int [task_stack_size]` for the *task* stack) or dynamic, if the user application supports malloc/alloc (**TNKernel** itself does not use dynamic memory allocation).

The *task_stack_size* value must to be chosen big enough to fit the *task_func* local variables and its switch context (processor registers, stack and instruction pointers, etc.).

The *task* stack must to be created as an array of `unsigned int`. Actually, the size of stack array element must be identical to the processor register size (for most 32-bits and 16-bits processors a register size equals `sizeof(int)`).

A parameter *task_stack_start* must point to the stack bottom. For instance, if the processor stack grows from the high to the low memory addresses and the task stack array is defined as `unsigned int xxx_xxx[task_stack_size]` (in C-language notation),
then the *task_stack_start* parameter has to be `&xxx_xxx[task_stack_size - 1]`.

tn_task_activate	Activate task after creation
tn_task_iactivate	Activate task after creation in interrupts

Function:

int tn_task_activate (TN_TCB * *task***)**

int tn_task_iactivate (TN_TCB * *task***)**

Parameter:

task	Pointer to the task TCB to be activated

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Input parameter(s) has a wrong value
TERR_OVERFLOW	Task is already active (not in DORMANT state)
TERR_NOEXS	Object is not a task or non-existent

Description:

These functions activate a task specified by the *task*. The task is moved from the DORMANT state to the READY state and the actions associated with task activation are performed.
If the task is not in the DORMANT state, a TERR_OVERFLOW error code is returned.

The **tn_task_iactivate()** function is a similar to the **tn_task_activate()** function, but has to be used in interrupts.

tn_task_terminate	Terminate task

Function:

int tn_task_terminate (TN_TCB * *task***)**

Parameter:

task	Pointer to the task TCB to be terminated

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Input parameter(s) has a wrong value
TERR_WSTATE	Task already terminated (i.e in DORMANT state) or attempt to terminate currently running task
TERR_WCONTEXT	Unacceptable system state for this request
TERR_NOEXS	Object is not a task or non-existent

Description:

This function terminates the task specified by the *task*. The *task* is moved to the DORMANT state.
When the task is waiting in a wait queue, the task is removed from the queue.
If activate requests exist (activation request count is 1) the count is decremented and the task is moved to the READY state. In this case the task starts execution from beginning (as after creation and activation), all mutexes locked by the task are unlocked etc. The task will have the lowest precedence among all tasks with the same priority in the READY state.

After termination, the task may be reactivated by the **tn_task_iactivate()** function or the **tn_task_activate()** function call.
In this case the task starts execution from the beginning (as after creation/activation).The task will have the lowest precedence among all tasks with the same priority in the READY state.

A task must not terminate itself by this function (use the **tn_task_exit()** function instead).
This function cannot be used in interrupts.

tn_task_delete Delete task

Function:

int tn_task_delete (TN_TCB * *task***)**

Parameter:

task	Pointer to the task TCB to be deleted

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Input parameter(s) has a wrong value
TERR_WCONTEXT	Unacceptable system's state for this request
TERR_NOEXS	Object is not a task or non-existent

Description:

This function deletes the task specified by the *task*. The *task* must to be in the DORMANT state, otherwise TERR_WCONTEXT will be returned.

This function resets the *id_task* field in the *task* structure to 0 and removes the task from the system tasks list. The *task* can not be reactivated after this function call (the *task* must be recreated).

This function cannot be invoked from interrupts.

tn_task_exit Terminate currently running task

Function:

void tn_task_exit (int *attr***)**

Parameter:

attr Exit option. Option values:

0
Currently running task will be terminated.

TN_EXIT_AND_DELETE_TASK

Currently running task will be terminated and then deleted

Description:

This function terminates the currently running task. The *task* is moved to the DORMANT state.
If activate requests exist (activation request count is 1) the count is decremented and the task is moved to the READY state. In this case the task starts execution from the beginning (as after creation and activation), all mutexes locked by the task are unlocked etc. The task will have the lowest precedence among all tasks with the same priority in the READY state.

After exiting, the task may be reactivated by the **tn_task_iactivate()** function or the **tn_task_activate()** function call.
In this case task starts execution from beginning (as after creation/activation).The task will have the lowest precedence among all tasks with the same priority in the READY state.

If this function is invoked with TN_EXIT_AND_DELETE_TASK parameter value, the task will be deleted after termination and cannot be reactivated (needs recreation).
This function cannot be invoked from interrupts.

tn_task_suspend	Suspend task

Function:

int tn_task_suspend (TN_TCB * *task***)**

Parameter:

task	Pointer to the task TCB to be suspended

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Input parameter(s) has a wrong value
TERR_OVERFLOW	Task already suspended
TERR_WSTATE	Task is not active (i.e in DORMANT state)
TERR_WCONTEXT	Unacceptable system state for this request
TERR_NOEXS	Object is not a task or non-existent

Description:

This function suspends the task specified by the *task*. If the task is runnable, it is moved to the SUSPENDED state. If the task is in the WAITING state, it is moved to the WAITING_SUSPENDED state.
A task can suspend itself.

tn_task_resume	Resume suspended task

Function:

int tn_task_resume (TN_TCB * *task***)**

Parameter:

task	Pointer to task TCB to be resumed

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Input parameter(s) has a wrong value
TERR_WSTATE	Task is not in SUSPEDED or WAITING_SUSPEDED state
TERR_NOEXS	Object is not a task or non-existent

Description:

This function releases the task specified by the *task* from the SUSPENDED state. If the *task* is in the SUSPEDED state, it is moved to the READY state, afterwards it has the lowest precedence among tasks with the same priority in the READY state.

If the *task* is in the WAITING_SUSPEDED state, it is moved to the WAITING state.

A task cannot resume itself.

tn_task_sleep Move currently running task in the sleep

Function:

int tn_task_sleep (unsigned int *timeout*)

Parameter:

timeout	Timeout value must be greater than 0. A value of TN_WAIT_INFINITE causes an infinite delay.

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Input parameter(s) has a wrong value
TERR_NOEXS	Object is not a task or non-existent

Description:

This function puts the currently running task to the sleep for at most *timeout* system ticks. When the timeout expires and the task was not suspended during the sleep, it is switched to runnable state. If the *timeout* value is TN_WAIT_INFINITE and the task was not suspended during the sleep, the task will sleep until another function call (like *tn_task_wakeup*() or similar) will make it runnable.

Each task has a wakeup request counter. If its value for currently running task is greater then 0, the counter is decremented by 1 and the currently running task is not switched to the sleeping mode and continues execution.

tn_task_wakeup	Wake up task from sleep
tn_task_iwakeup	Wake up task from sleep in interrupts

Function:

int tn_task_wakeup (TN_TCB * *task***)**

int tn_task_iwakeup (TN_TCB * *task***)**

Parameter:

task	Pointer to the task TCB to be wake up

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Input parameter(s) has a wrong value
TERR_OVERFLOW	Wakeup request already exists
TERR_WCONTEXT	Unacceptable system's state for this request
TERR_NOEXS	Object is not a task or non-existent

Description:

These functions wakes up the task specified by the *task* from sleep mode. The function placing the task into the sleep mode will return to the task without errors.
If the *task* is not in the sleep mode, the wakeup request for the *task* is queued and the wakeup request counter is incremented by 1.

The **tn_task_iwakeup()** function is a similar to the **tn_task_wakeup()** function, but has to be used in interrupts.

tn_task_release_wait	Release task from waiting or sleep
tn_task_irelease_wait	Release task from waiting or sleep in interrupts

Function:

int tn_task_release_wait (TN_TCB * *task***)**

int tn_task_irelease_wait (TN_TCB * *task***)**

Parameter:

task	Pointer to the task TCB to be released from waiting or sleep

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Input parameter(s) has a wrong value
TERR_WCONTEXT	Unacceptable system's state for function's request executing
TERR_NOEXS	Object is not a task or non-existent

Description:

These functions forcibly release the task specified by the *task* from waiting. If the *task* is in the WAITING state, it is moved to the READY state. If the *task* is in the WAITNG_SUSPENDED state, it is moved to the SUSPENDED state.

These functions release a *task* from any waiting state, including sleep mode. In last case, 0 is assigned to the wakeup request counter.

These functions do not cause a *task* in the SUSPENDED state to resume.

A task cannot specify itself in the *task* parameter.

The **tn_irelease_wait()** function is a similar to the **tn_task_release_wait()** function, but has to be used in interrupts.

Semaphore functions

Each semaphore has an associated data structure, defined (in file tn.h) as:

```
typedef struct _TN_SEM
{
   CDLL_QUEUE  wait_queue;
   int count;
   int max_count;
   int id_sem;
}TN_SEM;
```

In TN_SEM structure:

count	The resource availability (the number of unused resources).
wait_queue	A queue that manages the tasks waiting for the resources from the semaphore.
max_count	Max number of unused resources available to the semaphore.
id_sem	ID for verification (is it a semaphore or another object?) All semaphores have the same *id_sem* magic number (ver 2.x)

tn_sem_create   Create Semaphore

Function:

int tn_sem_create (TN_SEM * *sem***,**
int *start_value***,**
int *max_val* **)**

Parameters:

sem	Pointer to the semaphore TN_SEM structure to be created
start_value	The initial value of the resource counter after creation of the semaphore
max_val	The maximum resource counter value of the semaphore

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Input parameter(s) has a wrong value

Description:

This function creates a semaphore *sem*. A field *id_sem* of thestructure *sem* has to be set to 0 before the call of the function.

A memory for the *sem* must be allocated before its creation. An allocation may be static (global variables of the `TN_SEM` type for *sem)* or dynamic, if the user application supports malloc/alloc (**TNKernel** itself doesn't use dynamic memory allocation).

In **TNKernel** ver. 2.x, the semaphore's wait queue is always in the "first in - first out" order.

tn_sem_delete	Delete Semaphore

Function:

int tn_sem_delete (TN_SEM * *sem***)**

Parameters:

sem	Pointer to the semaphore TN_SEM structure to be deleted

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Input parameter(s) has a wrong value
TERR_NOEXS	Object is not a semaphore or non-existent

This function deletes a semaphore *sem*. All tasks that are waiting for the semaphore will be released from waiting with error code TERR_DLT.

The *id_sem* field of the structure *sem* is set to 0.

tn_sem_signal	Release Semaphore Resource
tn_sem_isignal	Release Semaphore Resource in interrupts

Function:

int tn_sem_signal (TN_SEM * *sem***)**

int tn_sem_isignal (TN_SEM * *sem***)**

Parameter:

sem	Pointer to the semaphore TN_SEM structure for the resource to be released

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Input parameter(s) has a wrong value
TERR_OVERFLOW	Semaphore Resource has *max_val* value
TERR_NOEXS	Object is not a semaphore or non-existent

Description:

These functions release one resource to the semaphore specified by the *sem*.
If any tasks are waiting for the semaphore, the task at the head of the associated wait queue is released from waiting, but the resource counter is not changed.
If there are no tasks waiting for the semaphore and the semaphore resource counter does not exceed the *max_val* of the semaphore, the semaphore resource counter is incremented by 1.

The **tn_sem_isignal()** function is a similar to the **tn_sem_signal( )** function, but has to be used in interrupts.

tn_sem_acquire Acquire Semaphore Resource

Function:

int tn_sem_acquire (TN_SEM * *sem*,
unsigned int *timeout* **)**

Parameters:

sem	Pointer to the semaphore TN_SEM structure of the resource to be acquired
timeout	Timeout value must be greater than 0. If *timeout* is TN_WAIT_INFINITE, the function's time-out interval never elapses.

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Input parameter(s) has a wrong value
TERR_TIMEOUT	Timeout expired
TERR_NOEXS	Object is not a semaphore or non-existent

Description:

This function acquires one resource from the semaphore specified by the *sem*.
If the resource counter of the semaphore is greater than 0, its resource counter is decremented by 1. In this case, the currently running task is not moved to the WAITING state.
If the resource counter of semaphore is 0, the currently running task is placed in the tail of the associated wait queue and moved to the WAITING state for the semaphore. In this case, the semaphore resource counter remains at value 0.
The value of the *timeout* is calculated in the system ticks.
When the timeout expires and the task wasn't suspended during the waiting, the task is switched to the runnable state.
If the value of timeout equals TN_WAIT_INFINITE, the wait never expires unless the semaphore is signalled.

tn_sem_polling	Acquire Semaphore Resource with polling
tn_sem_ipolling	Acquire Semaphore Resource in interrupts

Function:

int tn_sem_polling (TN_SEM * *sem***)**

int tn_sem_ipolling (TN_SEM * *sem***)**

Parameter:

sem	Pointer to the semaphore TN_SEM structure for the resource to be acquired

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Input parameter(s) has a wrong value
TERR_TIMEOUT	Resource counter's value is 0
TERR_NOEXS	Object is not a semaphore or non-existent

Description:

There functions use polling to acquire one resource from the semaphore specified by the *sem*. If the resource counter of the semaphore is greater than 0, it is decremented by 1.
If the resource count of the semaphore is 0, the function returns immediately with a TERR_TIMEOUT error code.

The **tn_sem_ipolling()** function is a similar to the **tn_sem_polling()** function, but has to be used in interrupts.

Mutex Functions

Each mutex has an associated data structure, defined (in file tn.h) as:

```
typedef struct _TN_MUTEX
{
   CDLL_QUEUE wait_queue;
   CDLL_QUEUE mutex_queue;
   CDLL_QUEUE lock_mutex_queue;
   int attr;

   TN_TCB * holder;
   int ceil_priority;
   int cnt;
   int id_mutex; //-- ID for verification(is it a mutex or another object?)
                 // All mutexes have the same id_mutex magic number (ver 2.x)
}TN_MUTEX;
```

In TN_MUTEX structure:

wait_queue	List of tasks that waits a mutex
mutex_queue	To include in the task's locked mutexes list (if any)
lock_mutex_queue	To include in the system's locked mutexes list
attr	Mutex creation attribute - TN_MUTEX_ATTR_CEILING or TN_MUTEX_ATTR_INHERIT
holder	Current mutex owner (task that locked mutex)
ceil_priority	Valid when the mutex was created with the TN_MUTEX_ATTR_CEILING attribute
cnt	Reserved
id_mutex	ID for verification (is it a mutex or another object). All mutexes have the same *id_mutex* magic number

tn_mutex_create	Create mutex

Function:

int tn_mutex_create(TN_MUTEX * *mutex*,
int *attribute*,
int *ceil_priority***)**

Parameters:

mutex	Pointer to already allocated TN_MUTEX structure of the mutex to be created
attribute	Creation attribute. Has to be one of: - TN_MUTEX_ATTR_INHERIT Mutex uses the priority inheritance protocol - TN_MUTEX_ATTR_CEILING – Mutex uses the priority ceiling protocol
ceil_priority	Valid only for the TN_MUTEX_ATTR_CEILING attribute. For the TN_MUTEX_ATTR_INHERIT attribute can have any value;

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Input parameter(s) has a wrong value

Description:

This function creates a mutex object. A field *id_mutex* of the structure *mutex* have to be set to 0 before the call of the function. A memory for the *mutex* must to be allocated before this function call. An allocation may be static (global variables with type **TN_MUTEX** for *mutex)* or dynamic, if the user application supports malloc/alloc (**TNKernel** itself does not use dynamic memory allocation).

The parameter *attribute* has to be TN_MUTEX_ATTR_INHERIT for the priority inheritance protocol or TN_MUTEX_ATTR_CEILING for the priority ceiling protocol.
The mutexes created with the TN_MUTEX_ATTR_INHERIT attribute ignore the *ceil_priority* parameter.

For the TN_MUTEX_ATTR_CEILING, the ceiling priority parameter should be set to the maximum priority of the task(s) that may lock the mutex.

tn_mutex_delete	Delete mutex

Function:

int tn_mutex_delete(TN_MUTEX * *mutex***)**

Parameters:

mutex	Pointer to already existing TN_MUTEX structure of mutex to be deleted

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Input parameter(s) has a wrong value
TERR_NOEXS	Object is not a mutex or non-existent

Description:

This function deletes a *mutex* object.
The current priority of the task locking the mutex will be changed, if it is necessary according to the priority inheritance or priority ceiling protocol.

The task that locked the mutex is not notified about the deletion of the mutex. Rather, it will receive an error TERR_DLT when it tries to unlock the mutex.
If there are tasks waiting to lock a mutex when it is deleted, they are released from waiting.

tn_mutex_lock	Lock Mutex
tn_mutex_lock_polling	Try to lock mutex

Function:

int tn_mutex_lock(TN_MUTEX * *mutex***,**
unsigned int *timeout* **)**

int tn_mutex_lock_polling(TN_MUTEX * *mutex***)**

Parameters:

mutex	Pointer to the mutex TN_MUTEX structure to be locked
timeout	Timeout value must be more than 0. If *timeout* is TN_WAIT_INFINITE, the function's time-out interval never elapses.

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Input parameter(s) has a wrong value
TERR_TIMEOUT	Timeout has been expired or, for function **tn_mutex_lock_polling()** only, the mutex is already locked
TERR_NOEXS	Object is not a mutex or non-existent
TERR_ILUSE	Illegal usage, e.g. trying to lock already locked mutex

Description:

These functions lock the *mutex*. If the *mutex* is not locked, a running task locks the mutex and returns without moving to the WAITING state.
If the *mutex* is locked, the **tn_mutex_lock()** function places the currently running task into the *mutex* wait queue and the task is moved to the the WAITING state for the *mutex*; the **tn_mutex_lock_polling()** function returns TERR_TIMEOUT error if the *mutex* is locked.

The value of *timeout* is calculated in system ticks.
When the *timeout* expires and the task wasn't suspended during the waiting, the task is switched to the runnable state.
A timeout TN_WAIT_INFINITE doesn't expire until the mutex can be locked.

If the running task already locked the *mutex* or has a base priority higher than the ceiling priority with a TN_MUTEX_ATTR_CEILING attributed mutex, these functions return a TERR_ILUSE error.

tn_mutex_unlock — Unlock Mutex

Function:

int tn_mutex_unlock(TN_MUTEX * *mutex* **)**

Parameters:

mutex	Pointer to the mutex TN_MUTEX structure to be unlocked

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Input parameter(s) has a wrong value
TERR_NOEXS	Object is not a mutex or non-existent
TERR_ILUSE	Illegal usage, eg. trying to unlock already unlocked mutex

Description:

This function unlocks the *mutex*.

If any tasks are waiting for the *mutex*, the task at the head of the *mutex* wait queue is released from waiting and locks the mutex. The current priority of this task will be changed if it is necessary according to the priority inheritance or the priority ceiling protocol.

If no task is waiting to lock the mutex, it goes to the unlocked state.

If the running task does not has the *mutex* locked, the functions returns a TERR_ILUSE error.

The current priority of the task unlocking the mutex will be changed if it is necessary according to the priority inheritance or the priority ceiling protocol.

Data Queue functions

Each data queue has an associated data structure defined (in file tn.h) as:

```
typedef struct _TN_DQUE
{
   CDLL_QUEUE  wait_send_list;
   CDLL_QUEUE  wait_receive_list;

   void ** data_fifo;
   int  num_entries;
   int  tail_cnt;
   int  header_cnt;
   int  id_dque;
}TN_DQUE;
```

In TN_DQUE structure:

wait_send_list	Wait queue for sending a data element
wait_receive_list	Wait queue for receiving a data element
data_fifo	Pointer to array of void* to store data queue data elements
num_entries	Capacity of *data_fifo* (max number of entries)
tail_cnt	Counter for processing data queue's *data_fifo*
header_cnt	Counter for processing data queue's *data_fifo*
id_dque	ID for verification (is it a data queue or another object?) All data queues have the same *id_dque* magic number (ver 2.x)

When the capacity of the data queue is set to zero (the *num_entries* is 0), a data queue can be used for tasks synchronization.
For instance, there are two tasks – the task A and the task B, and both tasks runs asynchronously.
If task A invokes *tn_queue_send()* first, the task A is moved to the WAITING state until task B calls *tn_queue_receive().*
If task B invokes *tn_queue_receive()* first, the task B is moved to the WAITING state until task A calls *tn_queue_send().*
When task A calls *tn_queue_send()* and task B calls *tn_queue_receive()*, the data transfer from task A to task B takes place and both tasks are moved to the runnable state.

tn_queue_create Create data queue

Function:

int tn_queue_create (TN_DQUE * *dque***,**
void ** *data_fifo***,**
int *num_entries* **)**

Parameters:

dque	Pointer to already allocated TN_DQUE structure of data queue to be created
data_fifo	Pointer to already existing array to store data queue entries. Each array element size equates **sizeof (void*)**. *data_fifo* can be NULL.
num_entries	Capacity of the data queue (max number of entries). Can be 0

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Input parameter(s) has a wrong value
TERR_NOEXS	Object is not a data queue or non-existent

Description:

This function creates a data queue. The field *id_dque* of the structure *dque* must be set to 0 before the call of the function. A memory for the *dque* and the *data_fifo* must to be allocated before the function call. An allocation may be static (global variables of type **TN_DQUE** for *dque* and void* *data_fifo* [*num_entries*] for *data_fifo*) or dynamic, if the user application supports malloc/alloc (**TNKernel** itself does not use dynamic memory allocation).

With the dynamic memory allocation, a size (in bytes) of *data_fifo* array has to be sizeof(void*) *x num_entries.*

tn_queue_delete	Delete the data queue

Function:

int tn_queue_delete (TN_DQUE * *dque***)**

Parameters:

dque	Pointer to TN_DQUE structure of data queue to be deleted

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Input parameter(s) has a wrong value
TERR_NOEXS	Object is not a data queue or non-existent

Description:

This function deletes the data queue, specified by the *dque.*
All tasks waiting for data queue resources will be released from the waiting with a error code TERR_DLT.
The field *id_dque* of structure *dque* will be set to 0.

tn_queue_send — Send (put) the data element to the data queue

Function:

int tn_queue_send (TN_DQUE * *dque***,**
void * *data_ptr***,**
unsigned int *timeout***)**

Parameters:

dque	Pointer to already allocated TN_DQUE structure of data queue to which the data element is send
data_ptr	Data element to be sent
timeout	Timeout value must be greater than 0. A value of TN_WAIT_INFINITE causes infinite waiting.

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Input parameter(s) has a wrong value
TERR_TIMEOUT	Timeout has been expired
TERR_NOEXS	Object is not a data queue or non-existent

Description:

This function sends the data element specified by the *data_ptr* to the data queue specified by the *dque*. If there are tasks in the data queue's wait_receive list already, the function releases the task from the head of the wait_receive list, makes this task runnable and transfers the parameter *data_ptr* to task's function, that caused it to wait..

If there are no tasks in the data queue's wait_receive list, parameter *data_ptr* is placed to the tail of data FIFO. If the data FIFO is full, the currently running task is switched to the waiting state and placed to the tail of data queue's send_receive list. If the *timeout* value is not a TN_WAIT_INFINITE, then after *timeout* expiration, function terminates immediately with a TERR_TIMEOUT error code.

tn_queue_send_polling	Send (put) the data element to the data queue with polling
tn_queue_isend_polling	Send (put) the data element to the data queue in interrupts

Function:

**int tn_queue_send_polling (TN_DQUE * *dque*,
void * data_ptr)**

**int tn_queue_isend_polling (TN_DQUE * *dque*,
void * data_ptr)**

Parameters:

dque	Pointer to already allocated TN_DQUE structure of data queue to which the data element is send
data_ptr	Data element to be sent

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Input parameter(s) has a wrong value
TERR_TIMEOUT	There are no free entries in data queue
TERR_NOEXS	Object is not a data queue or non-existent

Description:

The **tn_queue_send_polling()** function uses polling to send the data element specified by *data_ptr* to the data queue specified by *dque*.
If there are tasks in the data queue's wait_receive list already, the function releases the task from the head of the wait_receive list, makes this task runnable and transfers parameter *data_ptr* to the tasks function that caused it to wait.
If there is no room in the data FIFO, the function terminates immediately with a TERR_TIMEOUT error code.

The **tn_queue_isend_polling()** function is similar to the **tn_queue_send_polling()** function, but has to be used in interrupts.

tn_queue_receive	Receive (get) the data element from the data queue

Function:

int tn_queue_receive (TN_DQUE * *dque*,
void ** *data_ptr*,
unsigned int *timeout* **)**

Parameters:

dque	Pointer to already allocated TN_DQUE structure of data queue from which the data element is received
data_ptr	Address of pointer (type of void*) to receive data element from *dque*
timeout	Timeout value must be more than 0. A value of TN_WAIT_INFINITE causes infinite waiting.

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Input parameter(s) has a wrong value
TERR_TIMEOUT	Timeout has been expired
TERR_NOEXS	Object is not a data queue or non-existent

Description:

This function receives the data element from the data queue specified by the *dque* and places it into the address, specified by the *data_ptr*..

If the data FIFO already has entries, function removes an entry from the end of the data queue FIFO and returns it into the *data_ptr* function parameter. If there are task(s) in the data queue's wait_send list, the function removes the task from the head of wait_send list, makes this task runnable and puts the data entry, stored in this task, to the tail of data FIFO.

If there are no entries in the data FIFO and there are no tasks in the wait_send list, the currently running task is switched to waiting state and placed to the tail of the data queue's wait_receive list. If the *timeout* value is not TN_WAIT_INFINITE, then the function terminates immediately with TERR_TIMEOUT error code after *timeout* expiration.

tn_queue_receive_polling	Receive (get) the data element from the data queue with polling
tn_queue_ireceive	Receive (get) the data element from the data queue in interrupts

Function:

int tn_queue_receive_polling (TN_DQUE * *dque*,
void ** *data_ptr* **)**

int tn_queue_ireceive (TN_DQUE * *dque*,
void ** *data_ptr* **)**

Parameters:

dque	Pointer to TN_DQUE structure of data queue from which the data element is received
data_ptr	Address of pointer (type of void*) to receive data element from *dque*

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Input parameter(s) has a wrong value
TERR_TIMEOUT	There are no entries in data queue (data queue is empty)
TERR_NOEXS	Object is not a data queue or non-existent

Description:

The **tn_queue_receive_polling()** function uses polling to receive a data element from the data queue specified by the *dque* and place it into the address specified by the *data_ptr.*.
If the data FIFO already has entries, the function removes an entry from the end of the data queue FIFO and returns it into a *data_ptr* function's parameter. If there are task(s) in the data queue's wait_send list, the function removes the task from the head of the wait_send list, makes this task runnable and puts the data entry stored in this task TCB in the tail of the data FIFO.
If there are no entries in the data FIFO, the function terminates immediately with TERR_TIMEOUT error code.

The **tn_queue_ireceive()** function is similar to the **tn_queue_reveive_polling()** function, but has to be used in interrupts.

Eventflags functions

Each eventflag has an associated data structure defined (in file tn.h) as:

```
typedef struct _TN_EVENT
{
   CDLL_QUEUE wait_queue;
   int attr;
   unsigned int pattern;
   int id_event;
}TN_EVENT;
```

In TN_EVENT structure:

wait_queue	Wait queue for tasks waiting for an eventflag (this waiting will continue until every specified bit in the eventflag bit pattern is set).
attr	Eventflag attributes. Attributes are assigned to eventflag at the creation time (see the description of the **tn_event_create()** function).
pattern	Bit pattern with the state of eventflag's events
id_event	ID for verification (is it a evenflag or another object?) All eventflags have the same *id_event* magic number (ver 2.x)

tn_event_create Create the eventflag

Function:

int tn_event_create (TN_EVENT * *evf***,**
int *attr***,**
unsigned int *pattern***)**

Parameters:

evf	Pointer to already allocated TN_EVENT structure of eventflag to be created
attr	Eventflag attributes: TN_EVENT_ATTR_MULTI Multiple tasks are allowed to be in the waiting state for the eventflag TN_EVENT_ATTR_SINGLE Single task only is allowed to be in the waiting state for the eventflag TN_EVENT_ATTR_CLR (with TN_EVENT_ATTR_SINGLE only) Eventflag's entire bit pattern will be cleared when a task is released from the waiting state for the eventflag.
pattern	Initial value of the eventflag bit pattern

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Input parameter(s) has a wrong value

Description:

This function creates an eventflag specified by the *evf.* A field *id_evf* of the structure *evf* have to be set to 0 before the call of the function. A memory for the *evf* must to be allocated before the function call. An allocation may be static (global variable with type **TN_EVENT)** or dynamic, if the user application supports malloc/alloc (**TNKernel** itself does not use dynamic memory allocation).

The parameter *attr* must be TN_EVENT_ATTR_SINGLE or TN_EVENT_ATTR_MULTI.

If the eventflag has the TN_EVENT_ATTR_SINGLE attribute, it may also have TN_EVENT_ATTR_CLR.

An attributes TN_EVENT_ATTR_MULTI and TN_EVENT_ATTR_CLR are incompatible.

In **TNKernel** ver. 2.x, the eventflag's wait queue will be in the "first in -first out" order.

tn_event_delete	Delete the eventflag

Function:

int tn_event_delete (TN_EVENT * *evf***)**

Parameters:

evf	Pointer to TN_EVENT structure of eventflag to be deleted

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Input parameter(s) has a wrong value
TERR_NOEXS	Object is not a event flag or non-existent

Description:

This function deletes an eventflag specified by the *evf.*
All tasks waiting for the eventflag will be released with error code TERR_DLT.
The field *id_evf* of the structure *evf* will be set to 0.

tn_event_wait	Wait for eventflag
tn_event_wait_polling	Wait for eventflag with polling
tn_event_iwait	Wait for eventflag in interrupts

Function:

int tn_event_wait (TN_EVENT * *evf*,
unsigned int *wait_pattern*,
int *wait_mode*,
unsigned int * *p_flags_pattern*,
unsigned int *timeout***)**

int tn_event_wait_polling (TN_EVENT * *evf*,
unsigned int *wait_pattern*,
int *wait_mode*,
unsigned int * *p_flags_pattern***)**

int tn_event_iwait (TN_EVENT * *evf*,
unsigned int *wait_pattern*,
int *wait_mode*,
unsigned int * *p_flags_pattern***)**

Parameters:

evf	Pointer to TN_EVENT structure of eventflag to be wait
wait_pattern	Bit pattern to wait for. Cannot be 0.
wait_mode	Eventflag wait mode: TN_EVENT_WCOND_OR Any bit getting set is enough for release condition TN_EVENT_WCOND_AND Release condition requires all set bits matching
p_flags_pattern	Address of variable to receive pattern value after end of waiting
timeout	Timeout value must be greater than 0. A value of TN_WAIT_INFINITE causes infinite waiting.

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Input parameter(s) has a wrong value
TERR_ILUSE	Eventflag has been created with TN_EVENT_ATTR_SINGLE attribute and eventflag's wait queue is not empty
TERR_TIMEOUT	Timeout has expired - for **tn_event_wait( )** Release condition not satisfied - for **tn_event_iwait(**) and **tn_event_wait_polling( )**
TERR_NOEXS	Object is not a event flag or non-existent

Description:

The **tn_event_wait()** function causes currently running task to wait until the eventflag satisfies the release condition.

The release condition is determined by the bit pattern *wait_pattern* and the wait mode *wait_mode* parameters.

Once the release condition is satisfied, the bit pattern causing the release is returned through *p_flags_pattern*.
If the release condition is already satisfied when the **tn_event_wait()** is invoked, the function returns without causing the invoking task to wait. The eventflag bit pattern is still returned through *p_flags_pattern*.

If the eventflag *evf* has the TN_EVENT_ATTR_CLR attribute, all the bits in the eventflag's bit pattern are cleared.
If the release condition is not satisfied, a currently running task is placed in the eventflag's wait queue and switched to the WAITING state for the eventflag. If the *timeout* value is not TN_WAIT_INFINITE, then after *timeout* expiration the function terminates immediately with TERR_TIMEOUT error code.
If eventflag *evf* has the TN_EVENT_ATTR_SINGLE attribute and the eventflag's wait queue is not empty, the function returns with a TERR_ILUSE error code. This happens even if the release condition is already satisfied.
A parameter *wait_mode can be* specified as TN_EVENT_WCOND_OR or TN_EVENT_WCOND_AND.

If the parameter's value is TN_EVENT_WCOND_OR, any bit set is enough for the release condition.
If the parameter's value is TN_EVENT_WCOND_AND, the release condition requires all bits matching.

The **tn_event_wait_polling()** function is similar to the **tn_event_wait()** function, but uses polling to check release condition. If the release condition is not satisfied, **tn_event_wait_polling()** terminates immediately with a TERR_TIMEOUT error code.
The **tn_event_iwait()** function is similar to the **tn_event_wait_polling()** function, but has to be used in interrupts.

tn_event_set	Set eventflag
tn_event_iset	Set eventflag in interrupts

Function:

int tn_event_set (TN_EVENT * *evf*,
unsigned int *pattern***)**

int tn_event_iset (TN_EVENT * *evf*,
unsigned int *pattern***)**

Parameters:

Evf	Pointer to TN_EVENT structure of eventflag to be set
Pattern	Bit pattern to set. Cannot be 0.

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Input parameter(s) has a wrong value
TERR_NOEXS	Object is not a event flag or non-existent

Description:

These functions set the bits specified by the *pattern* in the eventflag specified by the *evf*. The set operation is a bitwise OR.

After the eventflag's bit pattern update action, any task(s) that satisfy their release conditions are released from waiting.

A multiple tasks can be released at once if the eventflag *evf* has the TN_EVENT_ATTR_MULTI attribute.

Next, if the eventflag *evf* has a TN_EVENT_ATTR_CLR attribute, the functions clear entire bit pattern and complete.

The **tn_event_iset()** function is similar to the **tn_event_set()** function, but has to be used in interrupts.

tn_event_clear	Clear eventflag
tn_event_iclear	Clear eventflag in interrupts

Function:

int tn_event_clear (TN_EVENT * *evf*,
unsigned int *pattern***)**

int tn_event_iclear (TN_EVENT * *evf*,
unsigned int *pattern***)**

Parameters:

Evf	Pointer to TN_EVENT structure of eventflag to be cleared
pattern	Bit pattern to clear. Cannot be 0xFFFFFFFF (all 1's).

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Input parameter(s) has a wrong value
TERR_NOEXS	Object is not a event flag or non-existent

Description:

This function clears the bits in the eventflag specified by the *evf* that correspond to 0 bit in the *pattern*. Bit pattern of the eventflag *evf* is updated by the bitwise AND operation with the value specified in *pattern*.

The **tn_event_iclear()** function is similar to the **tn_event_clear()** function, but has to be used in interrupts.

Fixed-sized memory pool functions

Each fixed-sized memory pool has an associated data structure, defined (in file tn.h) as:

```
typedef struct _TN_FMP
{
   CDLL_QUEUE wait_queue;
   unsigned int block_size;
   int num_blocks;
   void * start_addr;
   void * free_list;
   int fblkcnt;
   int id_fmp;
}TN_FMP;
```

In TN_FMP structure:

wait_queue	Wait queue for acquiring a memory block
block_size	Actual memory block size (in bytes)
num_blocks	Memory pool's capacity (actual max number fixed-sized memory blocks)
start_addr	Actual start address of memory pool storage area - memory, allocated to store memory blocks
free_list	Pointer to the free block list
blkcnt	Number of free blocks
id_fmp	ID for verification (is it a fixed-sized blocks memory pool or another object?). All fixed-sized blocks memory pool have the same *id_fmp* magic number (ver 2.x)

tn_fmem_create Create the fixed-sized memory pool

Function:

int tn_fmem_create (TN_FMP * *fmp*,
void * *start_addr*,
unsigned int *block_size*,
int *num_blocks* **)**

Parameters:

fmp	Pointer to the already allocated TN_FMP structure of fixed-sized memory pool to be created
start_addr	Start address of already allocated memory to store all memory blocks (memory pool area). Size of memory must be at least *block_size* * *num_blocks* (see more below).
block_size	Memory block size (in bytes)
num_blocks	Capacity (total number of memory blocks)

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Input parameter(s) has a wrong value

Description:

This function creates a fixed-sized memory pool. A field *id_fmp* of the structure *fmp* has to be set to 0 before the call of the function. A memory for the fixed-sized memory pool (pointed by the *start_addr)* and the TN_FMP structure *fmp* must be allocated before the function call.

An allocation may be static (global variables) or dynamic, if the user application supports malloc/alloc (**TNKernel** by itself doesn't use dynamic memory allocation).

For the best memory usage, *the block_size* value has to be aligned to the processor's memory alignment.

For instance, for the ARM processors the *block_size* value has to be 4,8,12…etc. bytes.

TNKernel has a special macro **MAKE_ALIG()** for this purpose.

In case of a static allocation, *start_addr* has to be, for instance:

```
unsigned int xxx_xxx[num_blocks * (MAKE_ALIG(block_size) / sizeof(int))];
start_addr = &xxx_xxx[0];
```

tn_fmem_delete	Delete the fixed-sized memory pool

Function:

int tn_fmem_delete (TN_FMP * *fmp*)

Parameters:

fmp	Pointer to already allocated TN_FMP structure of fixed-sized memory pool to be deleted

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Input parameter(s) has a wrong value
TERR_NOEXS	Object is not a fixed-sized memory pool or non-existent

Description:

This function deletes a fixed-sized memory pool specified by the *fmp.*
All tasks waiting for the fixed-sized memory pool resources will be released from the waiting with a TERR_DLT error code.
The *id_fmp* field of the structure *fmp* will be set to 0.

tn_fmem_get	Get fixed-sized memory block
tn_fmem_get_polling	Get fixed-sized memory block with polling
tn_fmem_get_ipolling	Get fixed-sized memory block in interrupts

Function:

int tn_fmem_get (TN_FMP * *fmp*,
void ** *p_data*,
unsigned int *timeout* **)**

int tn_fmem_get_polling (TN_FMP * *fmp*,
void ** *p_data* **)**

int tn_fmem_get_ipolling (TN_FMP * *fmp*,
void ** *p_data* **)**

Parameters:

fmp	Pointer to the TN_FMP structure of fixed-sized memory pool to get memory block
p_data	Address of the (void*) pointer to receive memory block's start address
timeout	Timeout value must be greater than 0. A value of TN_WAIT_INFINITE causes infinite waiting.

Return parameter:

TERR_NO_ERR	Normal completion	
TERR_WRONG_PARAM	Input parameter(s) has a wrong value	
TERR_TIMEOUT	Timeout has expired	- for **tn_fmem_get( )**
	There is no free memory block	- for **tn_fmem_get_polling( )** and **tn_fmem_get_ipolling( )**
TERR_NOEXS	Object is not a fixed-sized memory pool or non-existent	

Description:

The **tn_fmem_get()** function acquires a memory block from the fixed-sized memory pool. The start address of the memory block is returned through the *p_data*. The content of memory block is undefined.
When a free memory blocks are available in the memory pool area, one of the memory blocks is selected and takes on an acquired status.
If there are no memory blocks available, the invoking task is placed at the tail of the fixed-sized memory pool's wait queue and is moved to the WAITING state for a memory block.
If *timeout* value is not TN_WAIT_INFINITE, then after *timeout* expiration the function terminates immediately with a TERR_TIMEOUT error code.

The **tn_fmem_get_polling()** function is a similar to the **tn_fmem_get()** function**,** but uses polling to check availability of a free memory block.
If there is no free memory block, **tn_fmem_get_polling()** returns immediately with a TERR_TIMEOUT error code.
The **tn_fmem_get_ipolling()** function is a similar to the **tn_fmem_get_polling()** function, but has to be used in interrupts.

tn_fmem_release	Release fixed-sized memory block
tn_fmem_irelease	Release fixed-sized memory block in interrupts

Function:

int tn_fmem_release (TN_FMP * *fmp*,
void * *p_data* **)**

int tn_fmem_irelease (TN_FMP * *fmp*,
void * *p_data* **)**

Parameter:

fmp	Pointer to the TN_FMP structure of fixed-sized memory pool to release memory block
p_data	Start address of the memory block to be released

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Input parameter(s) has a wrong value
TERR_NOEXS	Object is not a fixed-sized memory pool or non-existent

Description:

These functions release the memory block starting from the address *p_data* to the fixed-sized memory pool specified by the *fmp*.
TNKernel does not checks the validity of the membership *p_data* in the *fmp.*

If a fixed-sized memory pool's wait queue is not empty, the task at the head of the wait queue acquires the released memory block and this task is released from waiting.

The **tn_fmem_irelease()** function is similar to the **tn_fmem_release()** function, but has to be used in interrupts.

References:

[1] L. Sha, R. Rajkumar, J. Lehoczky, **Priority Inheritance Protocols: An Approach to Real-Time Synchronization**, *IEEE Transactions on Computers, Vol.39, No.9, 1990*