
TNKernel

Real-Time Kernel

v. 1.0

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

Copyright © 2004 Yuri Tiomkin

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

TNKernel real time kernel

Copyright © 2004 Yuri Tiomkin
All rights reserved.

Permission to use, copy, modify, and distribute this software in source and binary forms and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation.

THIS SOFTWARE IS PROVIDED BY THE YURI TIOMKIN AND CONTRIBUTORS ``AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL YURI TIOMKIN OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Trademarks

Names mentioned in this manual may be trademarks of their respective companies. Brand and product names are trademarks or registered trademarks of their respective holders.

Document Version:

- 1.0

INTRODUCTION

TNKernel is compact and very fast real-time kernel for embedded 32/16 bits microprocessors. **TNKernel** performs preemptive priority-based scheduling with round-robin scheduling ability for tasks with identical priority.

Current version of **TNKernel** includes semaphores, data queues, event flags and fixed-sized memory pool operation. System's functions calls in interrupts are supported.

TNKernel is fully portable (written mostly on ANSI C except processor-specific parts), but current version of **TNKernel** has been ported for ARM microprocessors only.

TNKernel has been written "under the significant influence" of the μ ITRON 4.0 Specifications.

The μ ITRON 4.0 Specifications is an open real-time kernel specification developed by the ITRON Committee of the TRON Association. The μ ITRON 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 source code form free of charge under FreeBSD-like license.

TASKS

In **TNKernel**, task is branch of code that runs concurrently with another tasks from programmer's point of view. At the physical level, tasks are actually executed using processor's time sharing. Each task can be considered to be an independent program, which executes in its own context (processor registers, stack pointer, etc.).

When the currently running task loses its claim for executing (by the issue of a system call or interrupt), context switch is performed. The current context (processor registers, stack pointer, etc.) is saved and context of new task to run 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 order of task switching (execution) by using some rules. "Scheduler" is mechanism that controls the order of task execution.

TNKernel uses priority-based scheduling based on a priority level assigned to each task. The smaller the priority level value, the higher the priority level. **TNKernel** uses 32 levels of priority.

Priorities 0 (highest) and 31 (lowest) are reserved by system for internal using. 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 because a task with higher priority (sometimes same priority) is already executed. A task may execute at any time once the processor becomes available.

In **TNKernel**, both RUNNING state and READY state 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 task is waiting for these conditions. When a

task enters the WAIT/SUSPEND state, the task's context is saved. When the task resumes executing from the WAIT/SUSPEND state, the task's context is recovered.

WAIT/SUSPEND actually have one of three types:

- WAITING state

Task's execution is blocked until some synchronization action will occur (timeout expiration, semaphore, eventflag, etc.)

- SUSPENDED state

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

- WAITING_SUSPENDED state

Both WAITING and SUSPENDED states co-exists.

In **TNKernel**, if task leaves WAITING state, but SUSPENDED state exists, task is not switched to the READY/RUNNING state. Similarly, if task leaves SUSPENDED state, but WAITING state exists, task is not switched to the READY/RUNNING state. Task is switched to READY/RUNNING state only if there are no both WAITING and SUSPENDED states.

4. DORMANT state

The task has been initialized and task is not yet executing or task 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 tasks will execute unless the highest privilege task cannot be executed (for instance, placed in the WAITING state).

Among tasks with different priorities, the task with the highest priority is highest privilege task and will be executed.

Among tasks of the same priority, the task that entered into the runnable (RUNNING or READY) state earlier is highest privilege task and will be executed.

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 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 pre-determined 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 in interrupt(s), required context switching, system does it according to the architecture of processor (some processors uses different stack to service interrupts).

SYSTEM TASKS

In **TNKernel**, task with priority 0 (highest) uses for supporting system's ticks timer functionality (processing timeouts, delays, etc.) and task with priority 31 (lowest) uses for statistic purpose.

TNKernel automatically created these tasks at system's start.

TNKernel FUNCTIONALITY

1.Tasks

User may create tasks with priorities 1...30. User's task(s) should never communicate with tasks with priorities 0 and 31 (for instance, to make attempt to switch this task(s) into suspend state, etc.). System will reject any user's attempt to create task with priority 0 or 31.

More than one user's tasks can have the same (identical) priority. Tasks with identical priorities have ability for round-robin scheduling.

Task functions

(TNKernel version 1.x)

Function	Description
<i>tn_task_create</i>	Create task
<i>tn_task_activate</i>	Activate task after creation. Task switched from DORMANT state to runnable state
<i>tn_task_iactivate</i>	The same as above, but in interrupts
<i>tn_task_suspend</i>	Suspend task. If task is runnable, it is switched to the SUSPENDED state. If task in WAITING stage, it is moved into WAITING_SUSPENDED state
<i>tn_task_resume</i>	Resume suspended task - allows the task to continue its normal processing.
<i>tn_task_sleep</i>	Move currently running task into the sleep.
<i>tn_task_wakeup</i>	Wake up the task from sleep.
<i>tn_task_iwakeup</i>	The same as above, but in interrupts.
<i>tn_task_release_wait</i>	Forcibly release task from waiting (include sleep), but not from SUSPENDED state
<i>tn_task_irelease_wait</i>	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. Semaphore's wait queue manages the tasks waiting for resources from the semaphore. The resource counter is incremented by 1 when task releases a semaphore resource, and is decremented by 1 when 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's wait queue until a resource's arriving (another task returns it to the semaphore).

Semaphore functions

(TNKernel version 1.x)

Function	Description
<i>tn_sem_create</i>	Create semaphore
<i>tn_sem_signal</i>	Release semaphore resource
<i>tn_sem_issignal</i>	The same as above, but in interrupts
<i>tn_sem_acquire</i>	Acquire one resource from semaphore
<i>tn_sem_polling</i>	Acquire one resource from semaphore with polling
<i>tn_sem_ipolling</i>	The same as above, but in interrupts

3. Data Queues

A data queue is an FIFO that stores in each cell one pointer (type of *void**), called (in μ ITRON style) a data element.

A data queue also has an associated wait queues - 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 in FIFO. If there is no room in FIFO, the task is switched in WAITING state and placed in the data queue's *wait_send* queue until FIFO's room appears (another task gets data element from data queue).

A task that receiving a data element is tried to get a data element from the FIFO. If FIFO is empty (there are no data in the data queue), the task is switched in WAITING state and placed in the data queue's *wait_receive* queue until data element's arriving (another task put data element in data queue).

To use data queue just for synchronous message passing, size of FIFO shall to be 0.

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

Data Queue functions

(TNKernel version 1.x)

Function	Description
<i>tn_queue_create</i>	Create data queue
<i>tn_queue_send</i>	Send (put) the data element to the data queue
<i>tn_queue_send_polling</i>	Send (put) the data element to the data queue with polling
<i>tn_queue_isend_polling</i>	The same as above, but in interrupts
<i>tn_queue_receive</i>	Receive (get) the data element from the data queue
<i>tn_queue_receive_polling</i>	Receive (get) the data element from the data queue with polling
<i>tn_queue_ireceive</i>	The same as above, but in interrupts

4. Eventflags

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

A task may set specified bits when an event occurs and may clear specified bits when necessary. The task is waiting for events to occur will wait until every specified bit in the eventflag bit pattern is set. Tasks waiting for an eventflag are placed in the eventflag's wait queue.

Eventflag is very suitable sync object for cases when one task has to wait (for some reasons) many tasks, and vice versa, many tasks have to wait one task.

Eventflag functions

(TNKernel version 1.x)

Function	Description
<i>tn_event_create</i>	Create eventflag
<i>tn_event_wait</i>	Wait until eventflag satisfies the release condition
<i>tn_event_wait_polling</i>	Wait until eventflag satisfies the release condition with polling
<i>tn_event_await</i>	The same as above, but in interrupts
<i>tn_event_set</i>	Set eventflag
<i>tn_event_iset</i>	The same as above, but in interrupts
<i>tn_event_clear</i>	Clears the bits in the eventflag
<i>tn_event_iclear</i>	The same as above, but in interrupt

6. Fixed-Sized Memory Pools

A fixed-sized memory pool is used for dynamically managing fixed-sized memory blocks. A fixed-sized memory pool has a memory area where fixed-sized memory blocks are allocated and wait queue for acquiring a memory block.

If there are no free memory blocks, a task that trying to acquire a memory block will be placed in the wait queue until a free memory block's arriving (another task returns it to the memory pool).

Fixed-sized memory pool functions

(TNKernel version 1.x)

Function	Description
<i>tn_fmем_create</i>	Create Fixed-Sized Memory Pool
<i>tn_fmем_get</i>	Acquire (get) a memory block from memory pool
<i>tn_fmем_get_polling</i>	Acquire (get) a memory block from memory pool with polling
<i>tn_fmем_get_ipolling</i>	The same as above, but in interrupts
<i>tn_fmем_release</i>	Release (put back to pool) a memory block
<i>tn_fmем_irelease</i>	The same as above, but in interrupts

STARTING TNKernel

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

```
int main()
{
    //-- Operations before TNKernel's start (for instance, hardware
    initialization)
        .
        .
        .
    tn_start_system();    //-- Never returns
    return 0;             //-- not reachable
}
```

Function **tn_start_system()** performs all actions for initialization and start **TNKernel** (initializes system's global variables, creates tasks with priorities 0 and 31, calls start-up functions etc.)

Function **tn_start_system()** internally calls function **tn_app_init()**.

Contents (body) of function **tn_app_init()** has to be defined by user (may be empty). In this function user has to create all tasks, semaphores, data queues, memory pools etc., which user wants to have before system's start.

TNKernel TIME TICKS

For timeouts and delays calculation's purpose, **TNKernel** uses time ticks timer. In **TNKernel**, time ticks timer must to be kind of hardware timer, that produces interrupt for time ticks processing. Period of this timer is determined by user (usually in range 0.5...20 ms).

Within time ticks interrupt's processing, it is just necessarily to call functions **tn_tick_int_processing()** (see details below).

To minimize interrupt's processing time, **TNKernel** makes most time consumed processing inside task with priority 0. Function **tn_tick_int_processing()** releases task with priority 0 from sleep (see file tn.c).

ROUND ROBIN SCHEDULING IN TNKernel

TNKernel has ability to make round robin scheduling for the tasks with identical priority.

By default, round robin scheduling is turned off for all priorities. To enable round robin scheduling for tasks with certain priority level and to set time slice for the priority, user has to call function `tn_sys_tslice_ticks()`.

The time slice value is the same for all tasks with identical priority but may be different for each priority levels.

If round robin scheduling is enabled, each system's time tick interrupt increments the task's time slice counter for currently running task.

When the time slice interval is completed, the task is placed at the tail of the priority ready to run queue (this queue contents tasks in RUNNABLE state) and 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 switches to WAITING state (for instance, they are not acquires semaphore(s), etc.), tasks with priority less than priority of round-robin tasks will never run !

In most cases, there are no reasons to enable round robin scheduling. But in applications where multiple copies of the same code are to be run (GUI windows, etc.), round robin scheduling is an acceptable solution.

TNKernel PORT

There are few files in **TNKernel** files set, that have processor-depended contents:

tn_port.h

This file includes definitions for processor's memory alignment and macro for alignment's preparation.

tn_port.c

This file includes functions:

<i>tn_stack_init</i>	Creates task's stack frame. System invokes it at task's creation time.
<i>get_task_by_timer_queue</i>	Makes calculation for task's TCB start address by address of task's timer queue
<i>get_task_by_tsk_queue</i>	Makes calculation for task's TCB start address by address of task's wait queue

tn_arm_port.s

This file includes assembly-written functions:

Function	C-language prototype
<i>tn_switch_context</i>	void tn_switch_context (void)
<i>tn_cpu_irq_isr</i>	void tn_cpu_irq_isr (void)
<i>tn_cpu_save_sr</i>	int tn_cpu_save_sr (void)
<i>tn_cpu_restore_sr</i>	void tn_cpu_restore_sr (int sr)
<i>tn_start_exe</i>	void tn_start_exe (void)
<i>tn_chk_irq_disabled</i>	int tn_chk_irq_disabled (void)
<i>ffs_asm</i>	int ffs_asm (unsigned int val)
<i>tn_cpu_fiq_isr</i>	void tn_cpu_fiq_isr (void)

All assembly-written functions system uses for internal purpose only. There is no reason to invoke them from user's tasks.

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

Function **tn_cpu_irq_isr()** is used for the ARM processor's family. This function is invoked by ARM hardware to process IRQ interrupt.

Inside this function system calls function **tn_cpu_irq_handler()** to execute user's code for interrupt's processing. Then system checks context switch condition and (if it is necessarily) performs context switch within interrupts.

Function **tn_cpu_fiq_isr()** is used for ARM processor's family. This function is invoked by ARM hardware to process FIQ interrupt. It is similar to function **tn_cpu_irq_isr()**.

Function **tn_cpu_save_sr()** is used for the ARM processor's family. This function saves in additional variable contents of SPSR processor's register (it is necessarily in case of nested interrupts).

Function **tn_cpu_restore_sr()** is used for the ARM processor's family. This function restores contents of SPSR register that has been saved by function **tn_cpu_save_sr()** (it is necessarily in case of nested interrupts).

Function **tn_start_exe()** makes first context switching at system's start.

Function **tn_chk_irq_disabled()** is used for the ARM processor's family. This function checks the condition of CPSR processor's register for IRQ enabling/disabling status flag, and returns 1 if IRQ interrupt is disabled, otherwise returns 0.

Function **ffs_asm()** makes "find first setted bit" operation (start - from LSB). Function returns bit position (1...32), if bit with value '1' has been found, otherwise returns 0.

tn_user.c

This file includes functions:

- *tn_cpu_irq_handler*
- *tn_cpu_int_enable*

Function **tn_cpu_irq_handler()** is user's routine to processing IRQ and is used for the ARM processor's family. In this function user has to put his code to handle interrupts.

TNKernel invokes this function internally from **tn_cpu_irq_isr()** or **tn_cpu_fiq_isr()** without user's intervention (see above).

For ARM processor's family, **TNKernel** uses polling to recognize interrupt's sources. Discussion about strengths and weaknesses of this approach is out of scope for this document.

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

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

```
    .
}

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

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

Function **tn_cpu_int_enable()** enables all interrupts for vectors, utilized by user's project and then enables global interrupts. User must to enable system's time ticks interrupt in this function. **TNKernel** calls this function without user's intervention.

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

```
void tn_cpu_int_enable()
{
    //-- Enable UART interrupt
        .
        .
    //-- Enabled timer interrupt for time ticks (must).
        .
    //-- Enable DMA interrupt
        .
        .
        .
    //-- Enable global interrupts
        .
        .
        .
}
```

TNKernel API FUNCTIONS

System functions

tn_sys_tslice_ticks	Enable/disable priority's round robin scheduling
----------------------------	--

Function:

```
int tn_sys_tslice_ticks ( int priority,  
                        int value)
```

Parameter:

<i>priority</i>	Priority for which round-robin scheduling is enabled/disabled
<i>value</i>	Time slice value. Has to be more than 0 and less or equate MAX_TIME_SLICE. If <i>value</i> is NO_TIME_SLICE, round-robin scheduling for tasks with <i>priority</i> is disabled.

Return parameter:

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

Description:

This function controls round-robin scheduling for tasks with *priority*. Time slice *value* is calculated in system's ticks. The time slice *value* is the same for all tasks with identical *priority* but may be different for each priority levels.

Tasks functions

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

```
typedef struct _TN_TCB      //-- Version 1.x
{
    unsigned int * task_stk;    //-- Pointer to task's top of stack
    CDLL_QUEUE task_queue;    //-- Queue is used to include task in ready/wait lists
    CDLL_QUEUE timer_queue;    //-- Queue is used to include task in timer
                                //-- processing list (timeout, sleep, etc.)
    CDLL_QUEUE create_queue;    //-- Queue is used to include task in create list only

    int * stk_start;           //-- Base address of task's stack space
    int  stk_size;             //-- Task's stack size (in sizeof(void*),not bytes)

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

    int  await_pattern;        //-- Event wait pattern
    int  await_mode;           //-- Event wait mode: _AND or _OR

    void * data_elem;          //-- Store data queue entry (data element), if 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_createCreate 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:

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

Return parameter:

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

Description:

This function creates *task*. Memory for *task* and task stack must to be allocated before function's call. Allocation may be statical (global variables of TN_TCB type for *task* and `unsigned int [task_stack_size]` for task stack) or dynamical, if user's application supports malloc/alloc (TNKernel itself does not uses dynamic memory allocation).

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

Task stack must to be created as array of `unsigned int`. Actually, size of task stack array's element must be identical to the processor's register size (for most 32-bits and 16-bits processors register's size = `sizeof(int)`).

Parameter *task_stack_start* must point to stack bottom. For instance, if processor have stack growing from high memory addresses to low and task's stack array is defined as (in C-language notation) `unsigned int xxx_xxx [task_stack_size]`, then *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	Function's parameter(s) has a wrong value
TERR_OVERFLOW	Task is already active (not in DORMANT state)

Description:

These function(s) activates task specified by *task*. The task is moved from the DORMANT state to the READY state and the actions that must be taken at task activation time are performed.

If the task is not in the DORMANT state, a TERR_OVERFLOW error code is returned.

Function **tn_task_iactivate()** is similar to function **tn_task_activate()**, but has to be invoked in interrupts.

tn_task_suspendSuspend 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	Function's 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's state for function's request executing

Description:

This function suspends the task specified by *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.
Task can suspend itself.

tn_task_resumeResume suspend task

Function:

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

Parameter:

task Pointer to the task TCB to be resumed

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Function's parameter(s) has a wrong value
TERR_WSTATE	Task is not in SUSPEDED or WAITING_SUSPEDED state

Description:

This function releases the task specified by *task* from the SUSPENDED state. If the *task* is in the SUSPEDED state, it is moved to the READY state. After a *task* is moved from the SUSPENDED state to the READY state, the *task* has the lowest privilege 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_sleepMove currently running task in the sleep

Function:**int tn_task_sleep (unsigned int *timeout*)****Parameter:**

<i>timeout</i>	Timeout value must be more than 0. A value of TN_WAIT_INFINITE causes an infinite delay.
----------------	---

Return parameter:

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

Description:

This function moves the currently running task to the sleep for *timeout*. Value of *timeout* is calculated in system's ticks. When timeout is expired, task is switched to runnable state (if task didn't got SUSPEND state at sleep period). If *timeout* value is TN_WAIT_INFINITE, task will sleep until another function's call (like *tn_task_wakeup()* or similar) will makes it runnable (if task didn't got SUSPEND state at sleep period).

Each task has wakeup request counter. If wakeup request counter value for currently running task is more then 0, the counter is decremented by 1 and 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	Function's parameter(s) has a wrong value
TERR_OVERFLOW	Wakeup request already exists
TERR_WCONTEXT	Unacceptable system's state for function's request executing

Description:

These function(s) wakes up the task specified by *task* from sleep mode. Function that placed the task in the sleep mode will return to the task without errors.

If *task* is not in the sleep mode, the wakeup request for the *task* is queued and the wakeup request counter is incremented by 1.

Function **tn_task_iwakeup()** is similar to function **tn_task_wakeup()**, but has to be invoked 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	Function's parameter(s) has a wrong value
TERR_WCONTEXT	Unacceptable system's state for function's request executing

Description:

These function(s) forcibly releases the task specified by *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 SUSPEDED state.

These function(s) releases a *task* from any waiting state, include sleep mode. In last case, 0 is assigned to the wakeup request counter.

These function(s) do not cause a *task* in the SUSPENDED state to resume.

A task cannot specify itself in *task* parameter.

Function **tn_irelease_wait()** is similar to function **tn_task_release_wait()**, but has to be invoked in interrupts.

Semaphore functions

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

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

In TN_SEM structure:

<i>count</i>	The resource availability (the number of unused resources).
<i>wait_queue</i>	Queue that manages the tasks waiting for resources from the semaphore.
<i>max_count</i>	Max number of unused resources available to the semaphore.

tn_sem_createCreate Semaphore

Function:

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

Parameters:

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

Return parameter:

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

Description:

This function creates semaphore *sem*. Memory for *sem* must be allocated before function's call. Allocation may be statical (global variables of TN_SEM type for *sem*) or dynamical, if user's application supports malloc/alloc (TNKernel itself doesn't uses dynamic memory allocation).

In TNKernel ver. 1.x, the semaphore's wait queue always is in "first in - first out" order.

tn_sem_signal	Release Semaphore Resource
tn_sem_issignal	Release Semaphore Resource in interrupts

Function:

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

```
int tn_sem_issignal (TN_SEM * sem)
```

Parameter:

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

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Function's input parameter(s) has a wrong value
TERR_OVERFLOW	Semaphore Resource has <i>max_val</i> value

Description:

These function(s) releases one resource to the semaphore specified by *sem*. If any tasks are waiting for the *sem*, the task at the head of the *sem*'s wait queue is released from waiting. When this happens, the *sem* resource counter is not changed.

If there are no tasks are waiting for the *sem*, the semaphore resource counter is incremented by 1, if the counter value not exceeds the *max_val* of *sem*.

Function **tn_sem_issignal()** is similar to function **tn_sem_signal()**, but has to be invoked in interrupts.

tn_sem_acquire**Acquire Semaphore Resource**

Function:

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

Parameters:

<i>sem</i>	Pointer to the semaphore TN_SEM structure that resource to be acquire
<i>timeout</i>	Timeout value must be more than 0. A value of TN_WAIT_INFINITE makes acquire semaphore resource waiting process an infinite.

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Function's input parameter(s) has a wrong value
TERR_TIMEOUT	Timeout has been expired

Description:

This function acquires one resource from the semaphore specified by *sem*. If the resource counter of *sem* is more than 0, the *sem*'s resource counter is decremented by 1. In this case, the currently running task is not moved to the WAITING state.

If the resource count of *sem* is 0, the currently running task is placed in the in the tail of the *sem*'s wait queue and is moved to the waiting state for the *sem*. In this case, *sem*'s resource counter remains at value 0.

Value of *timeout* is calculated in system's ticks.

When timeout is expired (timeout TN_WAIT_INFINITE never expired), task is switched to runnable state (if task didn't got SUSPEND state at sleep period).

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 that resource to be acquire

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Function's input parameter(s) has a wrong value
TERR_TIMEOUT	Resource counter's value is 0

Description:

There function(s) uses polling to acquire one resource from the semaphore specified by *sem*. If the resource counter of *sem* is more than 0, the *sem*'s resource counter is decremented by 1. If the resource count of *sem* is 0, function terminates immediately with TERR_TIMEOUT error code.

Function **tn_sem_ipolling()** is similar to function **tn_sem_polling()**, but has to be invoked in interrupts.

Data Queue functions

Each data queue has 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;
}TN_DQUE;
```

In TN_DQUE structure:

<i>wait_send_list</i>	A wait queue for sending a data element
<i>wait_receive_list</i>	A wait queue for receiving a data element
<i>data_fifo</i>	Pointer to array of void* to store data queue entries (data elements)
<i>num_entries</i>	Capacity of <i>data_fifo</i> (max number of entries)
<i>tail_cnt</i>	Counter to processing data queue's <i>data_fifo</i>
<i>header_cnt</i>	Counter to processing data queue's <i>data_fifo</i>

When capacity of data queue is zero (*num_entries* is 0), data queue can be useable for tasks synchronization.

For instance, there are two tasks - task A and task B, and both tasks runs asynchronously.

If task A invokes *tn_queue_send()* first, task A is moved to the WAITING state until task B invokes *tn_queue_receive()*.

If task B invokes *tn_queue_receive()* first, task B is moved to the WAITING state until task A invokes *tn_queue_send()*.

When task A invokes *tn_queue_send()* and task B invokes *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_createCreate data queue

Function:

```
int tn_queue_create (TN_DQUE * dqe,  
                    void ** data_fifo,  
                    int num_entries )
```

Parameters:

<i>dqe</i>	Pointer to already allocated TN_DQUE structure of data queue to be created
<i>data_fifo</i>	Pointer to already existing array to store data queue entries. Array's element has size equated sizeof (void*) . <i>data_fifo</i> can be NULL.
<i>num_entries</i>	Capacity of data queue (max quantity of entries). Can be 0

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Function's input parameter(s) has a wrong value

Description:

This function creates data queue. Memory for *dqe* and *data_fifo* must to be allocated before function's call. Allocation may be statical (global variables with type **TN_DQUE** for *dqe* and `void* data_fifo [num_entries]` for *data_fifo*) or dynamical, if user's application supports malloc/alloc (TNKernel itself does not uses dynamic memory allocation).

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

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:

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

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Function's input parameter(s) has a wrong value
TERR_TIMEOUT	Timeout has been expired

Description:

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

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

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

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Function's input parameter(s) has a wrong value
TERR_TIMEOUT	There are no free entries in data queue

Description:

Function **tn_queue_send_polling()** uses polling to send the data element specified by *data_ptr* to the data queue specified by *dque*.

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

If there is no room in data FIFO, function terminates immediately with **TERR_TIMEOUT** error code.

Function **tn_queue_isend_polling()** is similar to function **tn_queue_send_polling()**, but has to be invoked 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:

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

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Function's input parameter(s) has a wrong value
TERR_TIMEOUT	Timeout has been expired

Description:

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

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

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

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:

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

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Function's input parameter(s) has a wrong value
TERR_TIMEOUT	There are no entries in data queue (data queue is empty)

Description:

Function **tn_queue_receive_polling()** uses polling to receive the data element from data queue specified by *dque* and place it by address, specified by *data_ptr*.

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

If there are no entries in the data FIFO, function terminates immediately with TERR_TIMEOUT error code.

Function **tn_queue_ireceive()** is similar to function **tn_queue_receive_polling()**, but has to be invoked in interrupts.

Eventflags functions

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

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

In TN_EVENT structure:

<i>wait_queue</i>	Wait queue for tasks, which are waiting for an eventflag (this waiting will continue until every specified bit in the eventflag bit pattern is set).
<i>attr</i>	Eventflag attribute(s). Attributes are assigned to eventflag at creation time (see tn_event_create() function's description).
<i>pattern</i>	Bit pattern with the state of eventflag's events

tn_event_createCreate the eventflag

Function:

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

Parameters:

<i>evf</i>	Pointer to already allocated TN_EVENT structure of eventflag to be created
<i>attr</i>	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.
<i>pattern</i>	Initial value of the eventflag bit pattern

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Function's input parameter(s) has a wrong value

Description:

This function creates a eventflag specified by *evf*. Memory for *evf* must be allocated before function's call. Allocation may be static (global variable with type **TN_EVENT**) or dynamic, if user's application supports malloc/alloc (TNKernel itself does not uses dynamic memory allocation).

Parameter *attr* must be TN_EVENT_ATTR_SINGLE or TN_EVENT_ATTR_MULTI.

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

Attributes TN_EVENT_ATTR_MULTI and TN_EVENT_ATTR_CLR are incompatible.

In TNKernel ver. 1.x, the eventflag's wait queue will be in "first in -first out" order.

<code>tn_event_wait</code>	Wait for eventflag
<code>tn_event_wait_polling</code>	Wait for eventflag with polling
<code>tn_event_await</code>	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_await (TN_EVENT * evf,
                  unsigned int wait_pattern,
                  int wait_mode,
                  unsigned int * p_flags_pattern)
```

Parameters:

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

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Function's 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 been expired - for <code>tn_event_wait()</code> Release condition is not satisfied - for <code>tn_event_await()</code> and <code>tn_event_wait_polling()</code>

Description:

Function `tn_event_wait()` 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 eventflag *evf* has `TN_EVENT_ATTR_CLR` attribute, all the bits in the eventflag's bit pattern are cleared. If the release condition is not satisfied, currently running task is placed in the eventflag's wait queue and is switched to the WAITING state for the eventflag. If *timeout* value is not `TN_WAIT_INFINITE`, then after *timeout* expiration function terminates immediately with `TERR_TIMEOUT` error code.

If eventflag *evf* has `TN_EVENT_ATTR_SINGLE` attribute and eventflag's wait queue is not empty, function returns with `TERR_ILUSE` error code. This happened even if the release condition is already satisfied.

Parameter *wait_mode* can be specified as `TN_EVENT_WCOND_OR` or `TN_EVENT_WCOND_AND`.

If parameter's value is `TN_EVENT_WCOND_OR`, any setted bit is enough for release condition.

If parameter's value is `TN_EVENT_WCOND_AND`, release condition is requires all setted bits matching.

Function **tn_event_wait_polling()** is similar to **tn_event_wait()**, but uses polling to check release condition. If the release condition is not satisfied, function **tn_event_wait_polling()** terminates immediately with `TERR_TIMEOUT` error code.

Function **tn_event_await()** is similar to function **tn_event_wait_polling()**, but has to be invoked in interrupts.

tn_event_set	Set eventflag
tn_event_isset	Set eventflag in interrupts

Function:

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

```
int tn_event_isset (TN_EVENT * evf,  
                   unsigned int pattern)
```

Parameters:

<i>Evf</i>	Pointer to TN_EVENT structure of eventflag to be set
<i>Pattern</i>	Bit pattern to set. Cannot be 0.

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Function's input parameter(s) has a wrong value

Description:

These function(s) sets the bits specified by *pattern* in the eventflag specified by *evf*. The set operation is bitwise OR.

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

Multiple tasks can be released at once if the eventflag *evf* has TN_EVENT_ATTR_MULTIPLE attribute.

Next, if the eventflag *evf* has TN_EVENT_ATTR_CLR attribute, function clears entire bit pattern and complete.

Function **tn_event_isset()** is similar to function **tn_event_set()**, but has to be invoked in interrupts.

<code>tn_event_clear</code>	Clear eventflag
<code>tn_event_iclear</code>	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:

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

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Function's input parameter(s) has a wrong value

Description:

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

Function `tn_event_iclear()` is similar to function `tn_event_clear()`, but has to be invoked in interrupts.

Fixed-sized memory pool functions

Each fixed-sized memory pool has 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;
}TN_FMP;
```

In TN_FMP structure:

<i>wait_queue</i>	Wait queue for acquiring a memory block
<i>block_size</i>	Actual memory block size (in bytes)
<i>num_blocks</i>	Memory pool's capacity (actual max number fixed-sized memory blocks)
<i>start_addr</i>	Actual start address of memory pool storage area - memory, allocated to store memory blocks
<i>free_list</i>	Pointer to free block list
<i>blkcnt</i>	Number of free blocks

tn_fmem_createCreate the fixed-sized memory pool

Function:

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

Parameters:

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

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Function's input parameter(s) has a wrong value

Description:

This function creates fixed-sized memory pool.

Memory for fixed-sized memory pool (pointed by *start_addr*) *task* and TN_FMP structure *fmp* must to be allocated before function's call.

Allocation may be static (global variables) or dynamical, if user's application supports malloc/alloc (TNKernel by itself doesn't uses dynamic memory allocation).

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

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

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

In case of 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_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:

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

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Function's input parameter(s) has a wrong value
TERR_TIMEOUT	Timeout has been expired - for tn_fmem_get() There is no free memory block - for tn_fmem_get_polling() and tn_fmem_get_ipolling()

Description:

Function **tn_fmem_get()** acquires a memory block from the fixed-sized memory pool.

The start address of the memory block is returned through *p_data*. Content of memory block is undefined.

When 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 fixed-sized memory pool's wait queue and is moved to the WAITING state for memory block.

If *timeout* value is not TN_WAIT_INFINITE, then after *timeout* expiration function terminates immediately with TERR_TIMEOUT error code.

Function **tn_fmem_get_polling()** is similar to **tn_fmem_get()**, but uses polling for check availability of free memory block.

If there is no free memory block, function **tn_fmem_get_polling()** terminates immediately with TERR_TIMEOUT error code.

Function **tn_fmem_get_ipolling()** is similar to function **tn_fmem_get_polling()**, but has to be invoked 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:

<i>fmp</i>	Pointer to TN_FMP structure of fixed-sized memory pool to release memory block
<i>p_data</i>	Start address of memory block to be released

Return parameter:

TERR_NO_ERR	Normal completion
TERR_WRONG_PARAM	Function's input parameter(s) has a wrong value

Description:

These function(s) releases the memory block starting from the address *p_data* to the fixed-sized memory pool specified by *fmp*.

TNKernel does not checks *p_data* for valid membership in *fmp*.

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

Function **tn_fmem_irelease()** is similar to function **tn_fmem_release()**,but has to be invoked in interrupts.