# TNeoKernel

## v1.0

Generated by Doxygen 1.8.8

Thu Oct 2 2014 03:07:03

# Contents

# Chapter 1

# TNeoKernel overview

TNeoKernel is a compact and fast real-time kernel for the embedded 32/16 bits microprocessors. It performs a preemptive priority-based scheduling and a round-robin scheduling for the tasks with identical priority.

TNeoKernel was born as a thorough review and re-implementation of TNKernel 2.7. The new kernel has well-formed code, inherited bugs are fixed as well as new features being added, it is well documented and tested carefully with unit-tests.

Currently it is available for PIC32 only, but will probably be ported to other architectures. Tested on PIC32MX.

API is changed somewhat, so it's not 100% compatible with TNKernel, hence the new name: TNeoKernel.

TNeoKernel is hosted at bitbucket: `http://bitbucket.org/dfrank/tneokernel`

Related pages:

- Foreword

- Quick guide

- Building the project

- PIC32 details

- Why reimplement TNKernel

- Differences from TNKernel API

- Unit tests

- Plans

- Legend

API reference:

- System services

- Tasks

- Mutexes

- Semaphores

- Fixed-memory blocks pool

- Event groups

- Data queues

# Chapter 2

# Foreword

Foreword.

This project was initially a fork of `PIC32 TNKernel port` by Anders Montonen. I don't like several design decisions of original TNKernel, as well as **many** of the implementation details, but Anders wants to keep his port as close to original TNKernel as possible. So I decided to fork it and have fun implementing what I want.

The more I get into how TNKernel works, the less I like its code. It appears as a very hastily-written project: there is a lot of code duplication and a lot of inconsistency, all of this leads to bugs. More, TNKernel is not documented well enough and there are no unit tests for it, so I decided to reimplement it almost completely. Refer to the page Why reimplement TNKernel for details.

I decided not to care much about compatibility with original TNKernel API because I really don't like several API decisions, so, I actually had to choose new name for this project, in order to avoid confusion, hence "TNeoKernel". Refer to the Differences from TNKernel API page for details.

Together with almost totally re-writing TNKernel, I've implemented detailed unit tests for it, to make sure I didn't break anything, and of course I've found several bugs in original TNKernel 2.7: refer to the section Bugs of TN↩Kernel 2.7. Unit tests are, or course, a "must-have" for the project like this; it's so strange bug original TNKernel seems untested.

Note that PIC32-dependent routines (such as context switch and so on) are originally implemented by Anders Montonen; I examined them in detail and changed several things which I believe should be implemented differently. Anders, great thanks for sharing your job.

Another existing PIC32 port, `the one by Alex Borisov`, also affected my project a bit. In fact, I used to use Alex's port for a long time, but it has several concepts that I don't like, so I had to move eventually. Nevertheless, Alex's port has several nice ideas and solutions, so I didn't hesitate to take what I like from his port. Alex, thank you too.

And, of course, great thanks to the author of original TNKernel, Yuri Tiomkin. Although the implementation of TN↩Kernel is far from perfect in my opinion, the ideas behind the implementation are generally really nice (that's why I decided to reimplement it instead of starting from scratch), and it was great entry point to the real-time kernels for me.

I would also like to thank my chiefs in the `ORION` company, Alexey Morozov and Alexey Gromov, for being flexible about my time.

# Chapter 3

# Quick guide

This page contains quick guide on system startup and important implementation details.

## 3.1 Time ticks

For the purpose of calculating timeouts, the kernel uses a time tick timer. In TNeoKernel, this time tick timer must to be a some kind of hardware timer that produces interrupt for time ticks processing. Throughout this text, this timer is referred to as *system timer*. The period of this timer is determined by user (typically 1 ms, but user is free to set different value). In the ISR for this timer, it is only necessary to call the `tn_tick_int_processing()` function:

```
//-- example for PIC32, hardware timer 5 interrupt:
tn_soft_isr(_TIMER_5_VECTOR)
{
   INTClearFlag(INT_T5);
   tn_tick_int_processing();
}
```

## 3.2 Starting the kernel

**Quick guide on startup process**

- You allocate arrays for idle task stack and interrupt stack, there is a convenience macro `TN_TASK_STA↩CK_DEF()` for that. It is good idea to consult the `TN_MIN_STACK_SIZE` to determine stack sizes (see example below).

- You provide callback function like `void init_task_create(void) { ... }`, in which at least one (and typically just one) your own task should be created and activated. This task should perform application initialization and create all the rest of tasks. See details in `TN_CBUserTaskCreate()`.

- You provide idle callback function to be called periodically from idle task. It's quite fine to leave it empty.

- In the `main()` you should:

  - disable interrupts globally by calling `tn_arch_int_dis()`;
  - perform some essential CPU configuration, such as oscillator settings and similar things.
  - setup *system timer* interrupt (from which `tn_tick_int_processing()` gets called)
  - perform any platform-dependent required actions (say, on PIC32 you should enable core software interrupt 0 with the lowest priority)
  - call `tn_sys_start()` providing all necessary information: pointers to stacks, their sizes and your callback functions.

- Kernel acts as follows:

    - performs all necessary housekeeping;

    - creates idle task;

    - calls your TN_CBUserTaskCreate() callback, in which your initial task is created with TN_TAS↩
    K_CREATE_OPT_START option;

    - performs first context switch (to your task with highest priority).

- At this point, system operates normally: your initial task gets executed and you can call whatever system
services you need. Typically, your initial task acts then as follows:

    - Perform initialization of various on-board peripherals (displays, flash memory chips, or whatever);

    - Initialize software modules used by application;

    - Create all the rest of your tasks (since everything is initialized already so that they can proceed with
    their job);

    - Eventually, perform its primary job (the job for which task was created at all).

**Basic example for PIC32**

This example project can be found in the TNeoKernel repository, in the examples/pic32/basic directory.

**Attention**

Before trying to build examples, please read Building the project page carefully: you need to copy con-
figuration file in the tneokernel directory to build it. Each example has tn_cfg_appl.h file, and you
should either create a symbolic link to this file from tneokernel/src/tn_cfg.h or just copy this file
as tneokernel/src/tn_cfg.h.

```c
/**
 * TNeoKernel PIC32 basic example
 */

/*******************************************************************************
 *    INCLUDED FILES
 ******************************************************************************/

#include <xc.h>
#include <plib.h>
#include <stdint.h>
#include "tn.h"




/*******************************************************************************
 *    PIC32 HARDWARE CONFIGURATION
 ******************************************************************************/

#pragma config FNOSC    = PRIPLL        // Oscillator Selection
#pragma config FPLLIDIV = DIV_4         // PLL Input Divider (PIC32 Starter Kit: use divide by 2 only)
#pragma config FPLLMUL  = MUL_20        // PLL Multiplier
#pragma config FPLLODIV = DIV_1         // PLL Output Divider
#pragma config FPBDIV   = DIV_2         // Peripheral Clock divisor
#pragma config FWDTEN   = OFF           // Watchdog Timer
#pragma config WDTPS    = PS1           // Watchdog Timer Postscale
#pragma config FCKSM    = CSDCMD        // Clock Switching & Fail Safe Clock Monitor
#pragma config OSCIOFNC = OFF           // CLKO Enable
#pragma config POSCMOD  = HS            // Primary Oscillator
#pragma config IESO     = OFF           // Internal/External Switch-over
#pragma config FSOSCEN  = OFF           // Secondary Oscillator Enable
#pragma config CP       = OFF           // Code Protect
#pragma config BWP      = OFF           // Boot Flash Write Protect
#pragma config PWP      = OFF           // Program Flash Write Protect
#pragma config ICESEL   = ICS_PGx2      // ICE/ICD Comm Channel Select
#pragma config DEBUG    = OFF           // Debugger Disabled for Starter Kit




/*******************************************************************************
 *    MACROS
 ******************************************************************************/
```

```
//-- instruction that causes debugger to halt
#define PIC32_SOFTWARE_BREAK()  __asm__ volatile ("sdbbp 0")

//-- system frequency
#define SYS_FREQ           80000000UL

//-- peripheral bus frequency
#define PB_FREQ            40000000UL

//-- kernel ticks (system timer) frequency
#define SYS_TMR_FREQ       1000

//-- system timer prescaler
#define SYS_TMR_PRESCALER           T5_PS_1_8
#define SYS_TMR_PRESCALER_VALUE     8

//-- system timer period (auto-calculated)
#define SYS_TMR_PERIOD              \
   (PB_FREQ / SYS_TMR_PRESCALER_VALUE / SYS_TMR_FREQ)




//-- idle task stack size, in words
#define IDLE_TASK_STACK_SIZE        (TN_MIN_STACK_SIZE + 16)

//-- interrupt stack size, in words
#define INTERRUPT_STACK_SIZE        (TN_MIN_STACK_SIZE + 64)

//-- stack sizes of user tasks
#define TASK_A_STK_SIZE    (TN_MIN_STACK_SIZE + 96)
#define TASK_B_STK_SIZE    (TN_MIN_STACK_SIZE + 96)
#define TASK_C_STK_SIZE    (TN_MIN_STACK_SIZE + 96)

//-- user task priorities
#define TASK_A_PRIORITY    7
#define TASK_B_PRIORITY    6
#define TASK_C_PRIORITY    5



/*******************************************************************************
 *    DATA
 ******************************************************************************/

//-- Allocate arrays for stacks: stack for idle task
//    and for interrupts are the requirement of the kernel;
//    others are application-dependent.
//
//    We use convenience macro TN_TASK_STACK_DEF() for that.

TN_TASK_STACK_DEF(idle_task_stack, IDLE_TASK_STACK_SIZE);
TN_TASK_STACK_DEF(interrupt_stack, INTERRUPT_STACK_SIZE);

TN_TASK_STACK_DEF(task_a_stack, TASK_A_STK_SIZE);
TN_TASK_STACK_DEF(task_b_stack, TASK_B_STK_SIZE);
TN_TASK_STACK_DEF(task_c_stack, TASK_C_STK_SIZE);



//-- task structures

struct TN_Task task_a = {};
struct TN_Task task_b = {};
struct TN_Task task_c = {};



/*******************************************************************************
 *    ISRs
 ******************************************************************************/

/**
 * system timer ISR
 */
tn_soft_isr(_TIMER_5_VECTOR)
{
   INTClearFlag(INT_T5);
   tn_tick_int_processing();
}



/*******************************************************************************
 *    FUNCTIONS
 ******************************************************************************/
```

```
void appl_init(void);

void task_a_body(void *par)
{
   //-- this is a first created application task, so it needs to perform
   //   all the application initialization.
   appl_init();

   //-- and then, let's get to the primary job of the task
   //   (job for which task was created at all)
   for(;;)
   {
      mPORTEToggleBits(BIT_0);
      tn_task_sleep(500);
   }
}

void task_b_body(void *par)
{
   for(;;)
   {
      mPORTEToggleBits(BIT_1);
      tn_task_sleep(1000);
   }
}

void task_c_body(void *par)
{
   for(;;)
   {
      mPORTEToggleBits(BIT_2);
      tn_task_sleep(1500);
   }
}

/**
 * Hardware init: called from main() with interrupts disabled
 */
void hw_init(void)
{
   SYSTEMConfig(SYS_FREQ, SYS_CFG_WAIT_STATES | SYS_CFG_PCACHE);

   //turn off ADC function for all pins
   AD1PCFG = 0xffffffff;

   //-- enable timer5 interrupt
   OpenTimer5((0
            | T5_ON
            | T5_IDLE_STOP
            | SYS_TMR_PRESCALER
            | T5_SOURCE_INT
            ),
         (SYS_TMR_PERIOD - 1)
         );

   //-- set timer5 interrupt priority to 2, enable it
   INTSetVectorPriority(INT_TIMER_5_VECTOR, INT_PRIORITY_LEVEL_2);
   INTSetVectorSubPriority(INT_TIMER_5_VECTOR, INT_SUB_PRIORITY_LEVEL_0);
   INTClearFlag(INT_T5);
   INTEnable(INT_T5, INT_ENABLED);

   //-- TNeoKernel PIC32 requirement:
   //   set up the software interrupt 0 with a priority of 1, subpriority 0
   //
   //   NOTE: the ISR is declared in kernel-provided file
   //   tn_arch_pic32_int_vec1.S, which should be included in the application
   //   project itself, in order to dispatch vector correctly.
   INTSetVectorPriority(INT_CORE_SOFTWARE_0_VECTOR, INT_PRIORITY_LEVEL_1);
   INTSetVectorSubPriority(INT_CORE_SOFTWARE_0_VECTOR, INT_SUB_PRIORITY_LEVEL_0);
   INTClearFlag(INT_CS0);
   INTEnable(INT_CS0, INT_ENABLED);

   //-- configure LED port pins
   mPORTESetPinsDigitalOut(BIT_0 | BIT_1 | BIT_2);
   mPORTEClearBits(BIT_0 | BIT_1 | BIT_2);

   //-- enable multi-vectored interrupt mode
   INTConfigureSystem(INT_SYSTEM_CONFIG_MULT_VECTOR);
}

/**
 * Application init: called from the first created application task
 */
void appl_init(void)
{
   //-- initialize various on-board peripherals, such as
```

```
   //   flash memory, displays, etc.
   //   (in this sample project there's nothing to init)

   //-- initialize various program modules
   //   (in this sample project there's nothing to init)


   //-- create all the rest application tasks
   tn_task_create(
         &task_b,
         task_b_body,
         TASK_B_PRIORITY,
         task_b_stack,
         TASK_B_STK_SIZE,
         NULL,
         (TN_TASK_CREATE_OPT_START)
         );

   tn_task_create(
         &task_c,
         task_c_body,
         TASK_C_PRIORITY,
         task_c_stack,
         TASK_C_STK_SIZE,
         NULL,
         (TN_TASK_CREATE_OPT_START)
         );
}

//-- idle callback that is called periodically from idle task
void idle_task_callback (void)
{
}

//-- create first application task(s)
void init_task_create(void)
{
   //-- task A performs complete application initialization,
   //   it's the first created application task
   tn_task_create(
         &task_a,                    //-- task structure
         task_a_body,                //-- task body function
         TASK_A_PRIORITY,            //-- task priority
         task_a_stack,               //-- task stack
         TASK_A_STK_SIZE,            //-- task stack size (in words)
         NULL,                       //-- task function parameter
         TN_TASK_CREATE_OPT_START    //-- creation option
         );

}

int32_t main(void)
{
#ifndef PIC32_STARTER_KIT
   /*The JTAG is on by default on POR.  A PIC32 Starter Kit uses the JTAG, but
     for other debug tool use, like ICD 3 and Real ICE, the JTAG should be off
     to free up the JTAG I/O */
   DDPCONbits.JTAGEN = 0;
#endif

   //-- unconditionally disable interrupts
   tn_arch_int_dis();

   //-- init hardware
   hw_init();

   //-- call to tn_sys_start() never returns
   tn_sys_start(
         idle_task_stack,
         IDLE_TASK_STACK_SIZE,
         interrupt_stack,
         INTERRUPT_STACK_SIZE,
         init_task_create,
         idle_task_callback
         );

   //-- unreachable
   return 1;
}

void __attribute__((naked, nomips16, noreturn)) _general_exception_handler(void)
{
   PIC32_SOFTWARE_BREAK();
   for (;;) ;
}
```

## 3.3   Round-robin scheduling

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_set()` 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.

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.

# Chapter 4

# Building the project

Some notes on building the project

## 4.1 Configuration file

TNeoKernel is intended to be built as a library, separately from main project (although nothing prevents you from bundling things together, if you want to).

There are various options available which affects API and behavior of the kernel. But these options are specific for particular project, and aren't related to the kernel itself, so we need to keep them separately.

To this end, file `tn.h` (the main kernel header file) includes `tn_cfg.h`, which isn't included in the repository (even more, it is added to `.hgignore` list actually). Instead, default configuration file `tn_cfg_default.h` is provided, and when you just cloned the repository, you might want to copy it as `tn_cfg.h`. Or even better, if your filesystem supports symbolic links, copy it somewhere to your main project's directory (so that you can add it to your VCS there), and create symlink to it named `tn_cfg.h` in the TNeoKernel source directory, like this:

```
$ cd /path/to/tneokernel/src
$ cp ./tn_cfg_default.h /path/to/main/project/lib_cfg/tn_cfg.h
$ ln -s /path/to/main/project/lib_cfg/tn_cfg.h ./tn_cfg.h
```

Default configuration file contains detailed comments, so you can read them and configure behavior as you like.

## 4.2 PIC32 port: MPLABX project

MPLABX project resides in the `src/arch/pic32/tneokernel.X` directory. This is a *library project* in terms of MPLABX, so if you use MPLABX you can easily add it to your main project by right-clicking `Libraries ->` `Add Library Project ....` Alternatively, of course you can just build it and use resulting `tneokernel.↩X.a` file in whatever way you like.

# Chapter 5

# PIC32 details

PIC32 port implementation details

## 5.1 Context switch

The context switch is implemented using the core software 0 interrupt. It should be configured to use the lowest priority in the system:

```
// set up the software interrupt 0 with a priority of 1, subpriority 0
INTSetVectorPriority(INT_CORE_SOFTWARE_0_VECTOR, INT_PRIORITY_LEVEL_1);
INTSetVectorSubPriority(INT_CORE_SOFTWARE_0_VECTOR, INT_SUB_PRIORITY_LEVEL_0);
INTEnable(INT_CS0, INT_ENABLED);
```

The interrupt priority level used by the context switch interrupt should not be configured to use shadow register sets.

**Attention**

> if tneokernel is built as a separate library, then the file `src/arch/pic32/tn_arch_pic32_int_←vec1.S` must be included in the main project itself, in order to dispatch vector1 (core software interrupt 0) correctly.

## 5.2 Interrupts

PIC32 port supports nested interrupts. The kernel provides C-language macros for calling C-language interrupt service routines, which can use either MIPS32 or MIPS16e mode. Both software and shadow register interrupt context saving is supported. Usage is as follows:

```
/* Timer 1 interrupt handler using software interrupt context saving */
tn_soft_isr(_TIMER_1_VECTOR)
{
   /* here is your ISR code, including clearing of interrupt flag, and so on */
}

/* High-priority UART interrupt handler using shadow register set */
tn_srs_isr(_UART_1_VECTOR)
{
   /* here is your ISR code, including clearing of interrupt flag, and so on */
}
```

**Attention**

> every ISR in your system should use kernel-provided macro, either `tn_soft_isr` or `tn_srs_isr`, instead of `__ISR` macro.

## 5.3  Interrupt stack

PIC32 uses a separate stack for interrupt handlers. Switching stack pointers is done automatically in the ISR handler wrapper macros. User should allocate array for interrupt stack and pass it as an argument to `tn_sys_start()`. Refer to the Starting the kernel section for the usage example and additional comments.

# Chapter 6

# Why reimplement TNKernel

Very brief explanation of the essential things I don't like in the TNKernel code.

## 6.1 One entry point, one exit point

The most common example that happens across all TNKernel sources is code like the following:

```
int my_function(void)
{
   tn_disable_interrupt();
   //-- do something

   if (error()){
      //-- do something

      tn_enable_interrupt();
      return ERROR;
   }
   //-- do something

   tn_enable_interrupt();
   return SUCCESS;
}
```

The code above is simplified; in the real TNKernel code, things are usually more complicated.

If you have multiple `return` statements or, even more, if you have to perform some action before return (`tn_↩ enable_interrupt()` in the example above), it's great job for `goto`:

```
int my_function(void)
{
   int ret = SUCCESS;
   tn_disable_interrupt();
   //-- do something

   if (error()){
      //-- do something
      ret = ERROR;
      goto out;
   }
   //-- do something

out:
   tn_enable_interrupt();
   return ret;
}
```

I understand there are a lot of people that don't agree with me on this (mostly because they religiously believe that `goto` is unconditionally evil), but anyway I decided to explain it. And yeah, original TNKernel code contains zero `goto` statement occurrences, but a lot of code like the one in first example above.

Needless to say, I already found such bug in original TNKernel 2.7 code. The function `tn_sys_tslice_↩ ticks()` looks as follows:

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

   TN_CHECK_NON_INT_CONTEXT

   tn_disable_interrupt();

   if(priority <= 0 || priority >= TN_NUM_PRIORITY-1 ||
                              value < 0 || value > MAX_TIME_SLICE)
      return TERR_WRONG_PARAM;

   tn_tslice_ticks[priority] = value;

   tn_enable_interrupt();
   return TERR_NO_ERR;
}
```

If you look closely, you can see that if wrong params were given, TERR_WRONG_PARAM is returned, and **interrupts remain disabled**. If we follow the *one entry point, one exit point* rule, this bug is much less likely to happen.

## 6.2 Don't repeat yourself

Original TNKernel 2.7 code has **a lot** of code duplication. So many similar things are done in several places just by copy-pasting the code.

- If we have similar functions (like, tn_queue_send(), tn_queue_send_polling() and tn_↩ queue_isend_polling()), the implementation is just copy-pasted, there's no effort to generalize things.

- Mutexes have complicated algorithms for task priorities. It is implemented in inconsistent, messy manner, which leads to bugs (refer to Bugs of TNKernel 2.7)

- Transitions between task states are done, again, in inconsistent copy-pasting manner. When we need to move task from, say, RUNNABLE state to the WAIT state, it's not enough to just clear one flag and set another one: we also need to remove it from whatever run queue the task is contained, probably find next task to run, then set reason of waiting, probably add to wait queue, set up timeout if specified, etc. In original TNKernel 2.7, there's no general mechanism to do this.

  Meanwhile, the correct way is to create three functions for each state:

  - to set the state;
  - to clear the state;
  - to test if the state active.

  And then, when we need to move task from one state to another, we typically should just call two functions: one for clearing current state, and one for settine a new one. It **is** consistent, and of course this approach is used in TNeoKernel.

As a result of the violation of the rule *Don't repeat yourself*, when we need to change something, we need to change it in several places. Needless to say, it is very error-prone practice, and of course there are bugs in original TNKernel because of that (refer to Bugs of TNKernel 2.7).

## 6.3 Macros that return from function

TNKernel uses architecture-depended macros like TN_CHECK_NON_INT_CONTEXT. This macro checks the current context (task or ISR), and if it is ISR, it returns TERR_WRONG_PARAM.

It isn't obvious to the reader of the code, but things like returning from function **must** be as obvious as possible.

It is better to invent some function that tests current context, and return the value explicitly:

```
enum TN_RCode my_function(void)
   enum TN_RCode rc = TN_RC_OK;

   // ...

   if (!tn_is_task_context()){
      rc = TN_RC_WCONTEXT;
      goto out;
   }

   // ...

out:
   return rc
}
```

## 6.4 Code for doubly-linked lists

TNKernel uses doubly-linked lists heavily, which is very good. I must admit that I really like the way data is organized in TNKernel. But, unfortunately, code that manages data is far from perfect, as I already mentioned.

So, let's get to the lists. I won't paste all the macros here, just make some overview. If we have a list, it's very common task to iterate through it. Typical snippet in TNKernel looks like this:

```
CDLL_QUEUE * curr_que;
TN_MUTEX * tmp_mutex;

curr_que = tn_curr_run_task->mutex_queue.next;
while(curr_que != &(tn_curr_run_task->mutex_queue))
{
   tmp_mutex = get_mutex_by_mutex_queque(curr_que);

   /* now, tmp_mutex points to the next object, so,
      we can do something useful with it */

   curr_que = curr_que->next;
}
```

This code is neither easy to read nor elegant. It's much better to use special macro for that (actually, similar macros are used across the whole Linux kernel code) :

```
TN_MUTEX * tmp_mutex;

tn_list_for_each_entry(tmp_mutex, &(tn_curr_run_task->mutex_queue), mutex_queue){
   /* now, tmp_mutex points to the next object, so,
      we can do something useful with it */
}
```

Much shorter and intuitive, isn't it? We even don't have to keep special `curr_que`.

## 6.5 Bugs of TNKernel 2.7

TNKernel 2.7 has several bugs, which are caught by detailed unit tests and fixed.

- We have two tasks: low-priority one `task_low` and high-priority one `task_high`. They use mutex `M1` with priority inheritance.

  - `task_low` locks `M1`
  - `task_high` tries to lock mutex `M1` and gets blocked -> priority of `task_low` elevates to the priority of `task_high`
  - `task_high` stops waiting for mutex by timeout -> priority of `task_low` remains elevated. The same happens if `task_high` is terminated by `tn_task_terminate()`.

- We have three tasks: two low-priority tasks `task_low1` and `task_low2`, and high-priority one `task_↵ high`. They use mutex `M1` with priority inheritance.

- **–** `task_low1` locks `M1`

- **–** `task_low2` tries to lock `M1` and gets blocked

- **–** `task_high` tries to lock `M1` and gets blocked -> priority if `task_low1` is elevated

- **–** `task_low1` unlocks `M1` ->

  - **\*** priority of `task_low1` returns to base value

  - **\*** `task_low2` locks `M1` because it's the next task in the mutex queue

  - **\*** now, priority of `task_low2` should be elevated, but it doesn't happen. Priority inversion is in effect.

- `tn_mutex_delete()` : if mutex is not locked, `TERR_ILUSE` is returned. Of course, task should be able to delete non-locked mutex;

- If task that waits for mutex is in `WAIT+SUSPEND` state, and mutex is deleted, `TERR_NO_ERR` is returned after returning from `SUSPEND` state, instead of `TERR_DLT`

- `tn_sys_tslice_ticks()` : if wrong params are given, `TERR_WRONG_PARAM` is returned and interrupts remain disabled.

- Probably not a "bug", but an issue in the data queue: actual capacity of the buffer is less by 1 than user has specified and allocated

Bugs with mutexes are the direct result of the inconsistency and copy-pasting the code, as well as lack of unit tests.

# Chapter 7

# Differences from TNKernel API

If you have experience of using TNKernel, you really want to read this.

## 7.1   Incompatible API changes

### 7.1.1   System startup

Original TNKernel code designed to be built together with main project only, there's no way to build as a separate library: at least, arrays for idle and timer task stacks are allocated statically, so size of them is defined at tnkernel compile time.

It's much better if we could pass these things to tnkernel at runtime, so, `tn_sys_start()` now takes pointers to stack arrays and their sizes. Refer to Starting the kernel section for the details.

### 7.1.2   Task creation API

In original TNKernel, one should give bottom address of the task stack to `tn_task_create()`, like this:

```
#define MY_STACK_SIZE   0x100
static unsigned int my_stack[ MY_STACK_SIZE ];

tn_task_create(/* ... several arguments omitted ... */
               &(my_stack[ MY_STACK_SIZE - 1]),
               /* ... several arguments omitted ... */);
```

Alex Borisov implemented it more conveniently in his port: one should give just array address, like this:

```
tn_task_create(/* ... several arguments omitted ... */
               my_stack,
               /* ... several arguments omitted ... */);
```

TNeoKernel uses the second way (i.e. the way used in port by Alex Borisov), and it does so independently of architecture.

### 7.1.3   Task wakeup count, activate count, suspend count

In original TNKernel, requesting non-sleeping task to wake up is quite legal and causes next call to `tn_task_sleep()` to not sleep. The same is with suspending/resuming tasks.

So, if you call `tn_task_wakeup()` on non-sleeping task first time, `TERR_NO_ERR` is returned. If you call it second time, before target task called `tn_task_sleep()`, `TERR_OVERFLOW` is returned.

All of this seems to me as a complete dirty hack, it probably might be used as a workaround to avoid race condition problems, or as a hacky replacement for semaphore.

It just encourages programmer to go with hacky approach, instead of creating straightforward semaphore and provide proper synchronization.

In TNeoKernel these "features" are removed, and if you try to wake up non-sleeping task, or try to resume non-suspended task, TN_RC_WSTATE is returned.

By the way, suspend_count is present in TCB structure, but is never used, so, it is just removed. And comments for wakeup_count, activate_count, suspend_count suggested that these fields are used for statistics, which is clearly not true.

### 7.1.4 Fixed memory pool: non-aligned address or block size

In original TNKernel it's illegal to pass block_size that is less than sizeof(int). But, it is legal to pass some value that isn't multiple of sizeof(int): in this case, block_size is silently rounded up, and therefore block_cnt is silently decremented to fit as many blocks of newly calculated block_size as possible. If resulting block_cnt is at least 2, it is assumed that everything is fine and we can go on.

Why I don't like it: firstly, silent behavior like this is generally bad practice that leads to hard-to-catch bugs. Secondly, it is inconsistency again: why is it legal for block_size not to be multiple of sizeof(int), but it is illegal for it to be less than sizeof(int)? After all, the latter is the partucular case of the former.

So, TNeoKernel returns TN_RC_WPARAM in these cases. User must provide start_addr and block_size that are properly aligned. As a generic rule, it is good practice to declare buffers for fmp like this:

```
#define MY_MEMORY_BUF_SIZE    8

struct MyMemoryItem {
   // ... arbitrary fields ...
};

int my_fmp_buf[
      MY_MEMORY_BUF_SIZE
        (TN_MAKE_ALIG_SIZE(sizeof(struct MyMemoryItem)) / sizeof(int))
      ];
struct TN_Fmp my_fmp;
```

And then, construct your my_fmp as follows:

```
tn_fmem_create( &my_fmp,
                my_fmp_buf,
                TN_MAKE_ALIG_SIZE(sizeof(struct MyMemoryItem)),
                MY_MEMORY_BUF_SIZE );
```

### 7.1.5 Task service return values cleaned

In original TNKernel, TERR_WCONTEXT is returned in the following cases:

- call to tn_task_terminate() for already terminated task;

- call to tn_task_delete() for non-terminated task;

- call to tn_task_change_priority() for terminated task;

- call to tn_task_wakeup()/tn_task_iwakeup() for terminated task;

- call to tn_task_release_wait()/tn_task_irelease_wait() for terminated task.

The actual error is, of course, wrong state, not wrong context; so, TNeoKernel returns TN_RC_WSTATE in these cases.

### 7.1.6 Force task releasing from wait

In original TNKernel, a call to `tn_task_release_wait()`/`tn_task_irelease_wait()` causes waiting task to wake up, regardless of wait reason, and `TERR_NO_ERR` is returned as a wait result. Actually I believe it is bad idea to ever use `tn_task_release_wait()`, but if we have this service, error code surely should be distinguishable from normal wait completion, so, new code is added: `TN_RC_FORCED`, and it is returned when task wakes up because of `tn_task_release_wait()` call.

### 7.1.7 Return code of tn_task_sleep()

In original TNKernel, `tn_task_sleep()` always returns `TERR_NO_ERR`, independently of what actually happened. In TNeoKernel, there are three possible return codes:

- `TN_RC_TIMEOUT` if timeout is actually in effect;

- `TN_RC_OK` if task was woken up by some other task with `tn_task_wakeup()`;

- `TN_RC_FORCED` if task was woken up forcibly by some other task with `tn_task_release_wait()`;

### 7.1.8 Events API is changed almost completely

In original TNKernel, I always found events API somewhat confusing. Why is this object named "event", but there are many flags inside, so that they can actually represent many events?

Meanwhile, attributes like `TN_EVENT_ATTR_SINGLE`, `TN_EVENT_ATTR_CLR` imply that "event" object is really just a single event, since it makes no sense to clear just **all** event bits when some particular event happened.

After all, when we call `tn_event_clear(&my_event_obj, flags)`, we might expect that `flags` argument actually specifies flags to clear. But in fact, we must invert it, to make it work: `~flags`. This is really confusing.

In TNeoKernel, there is no such *event* object. Instead, there is object *events group*. Attributes like `...SINGLE`, `...MULTI`, `...CLR` are removed, since they make no sense for events group. I have plans to offer a way to *connect* events group to queue and probably other kernel objects as well, so that queue will set and clear particular flag in the events group automatically, depending on whether a queue is empty. By means of that, it is quite easy to wait for data from multiple queues with just a single call to `tn_eventgrp_wait()`.

For detailed API reference, refer to the `tn_eventgrp.h`.

### 7.1.9 Zero timeout given to system functions

In original TNKernel, system functions refused to perform job and returned `TERR_WRONG_PARAM` if `timeout` is 0, but it is actually neither convenient nor intuitive: it is much better if the function behaves just like `...polling()` version of the function. All TNeoKernel system functions allows timeout to be zero: in this case, function doesn't wait.

## 7.2 Compatible API changes

### 7.2.1 Macro MAKE_ALIG()

There is a terrible mess with `MAKE_ALIG()` macro: TNKernel docs specify that the argument of it should be the size to align, but almost all ports, including original one, defined it so that it takes type, not size.

But the port by AlexB implemented it differently (i.e. accordingly to the docs) : it takes size as an argument.

When I was moving from the port by AlexB to another one, do you have any idea how much time it took me to figure out why do I have rare weird bug? :)

By the way, additional strange thing: why doesn't this macro have any prefix like `TN_`?

TNeoKernel provides macro `TN_MAKE_ALIG_SIZE()` whose argument is **size**, so, its usage is as follows: `TN↩
_MAKE_ALIG_SIZE(sizeof(struct MyStruct))`. This macro is preferred.

But for compatibility with messy `MAKE_ALIG()` from original TNKernel, there is an option `TN_API_MAKE_AL↩
IG_ARG` with two possible values;

- `TN_API_MAKE_ALIG_ARG__SIZE` - default value, use macro like this: `MAKE_ALIG(sizeof(struct
  my_struct))`, like in the port by Alex.

- `TN_API_MAKE_ALIG_ARG__TYPE` - use macro like this: `MAKE_ALIG(struct my_struct)`, like
  in any other port.

By the way, I wrote to the author of TNKernel (Yuri Tiomkin) about this mess, but he didn't answer anything. It's a
pity of course, but we have what we have.

### 7.2.2   Recursive mutexes

Sometimes I feel lack of mutexes that allow recursive locking. Yeah I know there are developers who believe that
recursive locking leads to the code of lower quality, and I understand it. Even Linux kernel doesn't have recursive
mutexes.

Sometimes they are really useful though (say, if you want to use some third-party library that requires locking
primitives to be recursive), so I decided to implement an option for that: `TN_MUTEX_REC`. If it is non-zero, mutexes
allow recursive locking; otherwise you get `TN_RC_ILLEGAL_USE` when you try to lock mutex that is already
locked by this task. Default value: `1`.

### 7.2.3   Convenience macros for stack arrays definition

You can still use "manual" definition of stack arrays, like that:

```
TN_ARCH_STK_ATTR_BEFORE
TN_UWord my_task_stack[ MY_TASK_STACK_SIZE ]
TN_ARCH_STK_ATTR_AFTER;
```

Although it is recommended to use convenience macro for that: `TN_TASK_STACK_DEF()`. See `tn_task_↩
create()` for the usage example.

### 7.2.4   Convenience macros for fixed memory block pool buffers definition

Similarly to the previous section, you can still use "manual" definition of the buffer for fixed memory block pool, it
is recommended to use convenience macro for that: `TN_FMEM_BUF_DEF()`. See `tn_fmem_create()` for
usage example.

### 7.2.5   Things renamed

In original TNKernel thare is a lot of inconsistency with naming stuff:

- Why do we have `tn_queue_send_polling()` / `tn_queue_isend_polling()` (notice the `i`
  letter before the verb, not before `polling`), but `tn_fmem_get_polling()` / `tn_fmem_get_↩
  ipolling()` (notice the `i` letter before `polling`)?

- All the system service names follow the naming scheme `tn_<noun>_<verb>[_<adjustment>]()`,
  but the `tn_start_system()` is special, for some strange reason. To make it consistent, it should be
  named `tn_system_start()` or `tn_sys_start()`;

- A lot of macros don't have `TN_` prefix;

- etc

So, a lot of things (functions, macros, etc) has renamed. Old names are also available through `tn_↩ oldsymbols.h`, which is included automatically if `TN_OLD_TNKERNEL_NAMES` option is non-zero.

## 7.3 Changes that do not affect API directly

### 7.3.1 No timer task

Yes, timer task's job is important: it manages `tn_wait_timeout_list`, i.e. it wakes up tasks whose timeout is expired. But it's actually better to do it right in `tn_tick_int_processing()` that is called from timer ISR, because presence of the special task provides significant overhead. Look at what happens when timer interrupt is fired (assume we don't use shadow register set for that, which is almost always the case):

(measurements were made at PIC32 port)

- Current context (23 words) is saved to the interrupt stack;

- ISR called: particularly, `tn_tick_int_processing()` is called;

- `tn_tick_int_processing()` disables interrupts, manages round-robin (if needed), then it wakes up `tn_timer_task`, sets `tn_next_task_to_run`, and enables interrupts back;

- `tn_tick_int_processing()` finishes, so ISR macro checks that `tn_next_task_to_run` is different from `tn_curr_run_task`, and sets `CS0` interrupt bit, so that context should be switched as soon as possible;

- Context (23 words) gets restored to whatever task we interrupted;

- `CS0` ISR is immediately called, so full context (32 words) gets saved on task's stack, and context of `tn_↩ timer_task` is restored;

- `tn_timer_task` disables interrupts, performs its not so big job (manages `tn_wait_timeout_list`), puts itself to wait, enables interrupts and pends context switching again;

- `CS0` ISR is immediately called, so full context of `tn_timer_task` gets saved in its stack, and then, after all, context of my own interrupted task gets restored and my task continues to run.

I've measured with MPLABX's stopwatch how much time it takes: with just three tasks (idle task, timer task, my own task with priority 6), i.e. without any sleeping tasks, all this routine takes **682 cycles**. So I tried to get rid of `tn_timer_task` and perform its job right in the `tn_tick_int_processing()`.

Previously, application callback was called from timer task; since it is removed now, startup routine has changed, refer to Starting the kernel for details.

Now, the following steps are performed when timer interrupt is fired:

- Current context (23 words) is saved to the interrupt stack;

- ISR called: particularly, `tn_tick_int_processing()` is called;

- `tn_tick_int_processing()` disables interrupts, manages round-robin (if needed), manages `tn_↩ wait_timeout_list`, and enables interrupts back;

- `tn_tick_int_processing()` finishes, ISR macro checks that `tn_next_task_to_run` is the same as `tn_curr_run_task`

- Context (23 words) gets restored to whatever task we interrupted;

That's all. It takes **251 cycles**: 2.7 times less.

So, we need to make sure that interrupt stack size is enough for this (not big) job. As a result, RAM is saved (since you don't need to allocate stack for timer task) and things work much faster. Win-win.

---

# Chapter 8

# Unit tests

Brief information on the implementation of unit tests

## 8.1 How tests are implemented

Briefly: there is a high-priority task like "test director", which creates worker tasks as well as various kernel objects (queues, mutexes, etc), and then orders to workers, like:

- Task A, you lock the mutex M1

- Task B, you lock the mutex M1

- Task C, you lock the mutex M1

- Task A, you delete the mutex M1

After each step it waits for workers to complete their job, and then checks if things are as expected: task states, task priorities, last return values of services, various properties of objects, etc.

Detailed log is written to the UART. Typically, for each step, the following is written:

- verbatim comment is written,

- director writes what does it do,

- each worker writes what does it do,

- director checks things and writes detailed report.

Of course there is a mechanism for writing such scenarios. Here is a part of code that specifies the sequence with locking and deleting mutex explained above:

```
TNT_TEST_COMMENT("A locks M1");
TNT_ITEM__SEND_CMD_MUTEX(TNT_TASK__A, MUTEX_LOCK, TNT_MUTEX__1);
TNT_ITEM__WAIT_AND_CHECK_DIFF(
      TNT_CHECK__MUTEX(TNT_MUTEX__1, HOLDER, TNT_TASK__A);
      TNT_CHECK__MUTEX(TNT_MUTEX__1, LOCK_CNT, 1);

      TNT_CHECK__TASK(TNT_TASK__A, LAST_RETVAL, TN_RC_OK);
      );

TNT_TEST_COMMENT("B tries to lock M1 -> B blocks, A has priority of B");
TNT_ITEM__SEND_CMD_MUTEX(TNT_TASK__B, MUTEX_LOCK, TNT_MUTEX__1);
TNT_ITEM__WAIT_AND_CHECK_DIFF(
      TNT_CHECK__TASK(TNT_TASK__B, LAST_RETVAL, TWORKER_MAN__LAST_RETVAL__UNKNOWN);
      TNT_CHECK__TASK(TNT_TASK__B, WAIT_REASON, TSK_WAIT_REASON_MUTEX_I);

      TNT_CHECK__TASK(TNT_TASK__A, PRIORITY, priority_task_b);
      );
```

```
TNT_TEST_COMMENT("C tries to lock M1 -> C blocks, A has priority of C");
TNT_ITEM__SEND_CMD_MUTEX(TNT_TASK__C, MUTEX_LOCK, TNT_MUTEX__1);
TNT_ITEM__WAIT_AND_CHECK_DIFF(
      TNT_CHECK__TASK(TNT_TASK__C, LAST_RETVAL, TWORKER_MAN__LAST_RETVAL__UNKNOWN);
      TNT_CHECK__TASK(TNT_TASK__C, WAIT_REASON, TSK_WAIT_REASON_MUTEX_I);

      TNT_CHECK__TASK(TNT_TASK__A, PRIORITY, priority_task_c);
      );

TNT_TEST_COMMENT("A deleted M1 -> B and C become runnable and have retval TN_RC_DELETED, A has its base
      priority");
TNT_ITEM__SEND_CMD_MUTEX(TNT_TASK__A, MUTEX_DELETE, TNT_MUTEX__1);
TNT_ITEM__WAIT_AND_CHECK_DIFF(
      TNT_CHECK__TASK(TNT_TASK__B, LAST_RETVAL, TN_RC_DELETED);
      TNT_CHECK__TASK(TNT_TASK__C, LAST_RETVAL, TN_RC_DELETED);
      TNT_CHECK__TASK(TNT_TASK__B, WAIT_REASON, TSK_WAIT_REASON_DQUE_WRECEIVE)
      ;
      TNT_CHECK__TASK(TNT_TASK__C, WAIT_REASON, TSK_WAIT_REASON_DQUE_WRECEIVE)
      ;

      TNT_CHECK__TASK(TNT_TASK__A, PRIORITY, priority_task_a);

      TNT_CHECK__MUTEX(TNT_MUTEX__1, HOLDER, TNT_TASK__NONE);
      TNT_CHECK__MUTEX(TNT_MUTEX__1, LOCK_CNT, 0);
      TNT_CHECK__MUTEX(TNT_MUTEX__1, EXISTS, 0);
      );
```

And here is the appropriate part of log that is echoed to the UART:

```
//-- A locks M1 (line 404 in ../source/appl/appl_tntest/appl_tntest_mutex.c)
[I]: tnt_item_proceed():2101: ----- Command to task A: lock mutex M1 (0xa0004c40)
[I]: tnt_item_proceed():2160: Wait 80 ticks
[I]: [Task A]: locking mutex (0xa0004c40)..
[I]: [Task A]: mutex (0xa0004c40) locked
[I]: [Task A]: waiting for command..
[I]: tnt_item_proceed():2178: Checking:
[I]: * Task A: priority=6 (as expected), wait_reason=DQUE_WRECEIVE (as expected), last_retval=
      TN_RC_OK (as expected)
[I]: * Task B: priority=5 (as expected), wait_reason=DQUE_WRECEIVE (as expected), last_retval=NOT-YET-
      RECEIVED (as expected)
[I]: * Task C: priority=4 (as expected), wait_reason=DQUE_WRECEIVE (as expected), last_retval=NOT-YET-
      RECEIVED (as expected)
[I]: * Mutex M1: holder=A (as expected), lock_cnt=1 (as expected), exists=yes (as expected)

//-- B tries to lock M1 -> B blocks, A has priority of B (line 413 in
      ../source/appl/appl_tntest/appl_tntest_mutex.c)
[I]: tnt_item_proceed():2101: ----- Command to task B: lock mutex M1 (0xa0004c40)
[I]: tnt_item_proceed():2160: Wait 80 ticks
[I]: [Task B]: locking mutex (0xa0004c40)..
[I]: tnt_item_proceed():2178: Checking:
[I]: * Task A: priority=5 (as expected), wait_reason=DQUE_WRECEIVE (as expected), last_retval=
      TN_RC_OK (as expected)
[I]: * Task B: priority=5 (as expected), wait_reason=MUTEX_I (as expected), last_retval=NOT-YET-RECEIVED (
      as expected)
[I]: * Task C: priority=4 (as expected), wait_reason=DQUE_WRECEIVE (as expected), last_retval=NOT-YET-
      RECEIVED (as expected)
[I]: * Mutex M1: holder=A (as expected), lock_cnt=1 (as expected), exists=yes (as expected)

//-- C tries to lock M1 -> B blocks, A has priority of C (line 422 in
      ../source/appl/appl_tntest/appl_tntest_mutex.c)
[I]: tnt_item_proceed():2101: ----- Command to task C: lock mutex M1 (0xa0004c40)
[I]: tnt_item_proceed():2160: Wait 80 ticks
[I]: [Task C]: locking mutex (0xa0004c40)..
[I]: tnt_item_proceed():2178: Checking:
[I]: * Task A: priority=4 (as expected), wait_reason=DQUE_WRECEIVE (as expected), last_retval=
      TN_RC_OK (as expected)
[I]: * Task B: priority=5 (as expected), wait_reason=MUTEX_I (as expected), last_retval=NOT-YET-RECEIVED (
      as expected)
[I]: * Task C: priority=4 (as expected), wait_reason=MUTEX_I (as expected), last_retval=NOT-YET-RECEIVED (
      as expected)
[I]: * Mutex M1: holder=A (as expected), lock_cnt=1 (as expected), exists=yes (as expected)

//-- A deleted M1 -> B and C become runnable and have retval TN_RC_DELETED, A has its base priority (line
      431 in ../source/appl/appl_tntest/appl_tntest_mutex.c)
[I]: tnt_item_proceed():2101: ----- Command to task A: delete mutex M1 (0xa0004c40)
[I]: tnt_item_proceed():2160: Wait 80 ticks
[I]: [Task A]: deleting mutex (0xa0004c40)..
[I]: [Task C]: mutex (0xa0004c40) locking failed with err=-8
[I]: [Task C]: waiting for command..
[I]: [Task B]: mutex (0xa0004c40) locking failed with err=-8
[I]: [Task B]: waiting for command..
[I]: [Task A]: mutex (0xa0004c40) deleted
[I]: [Task A]: waiting for command..
[I]: tnt_item_proceed():2178: Checking:
```

```
[I]: * Task A: priority=6 (as expected), wait_reason=DQUE_WRECEIVE (as expected), last_retval=
       TN_RC_OK (as expected)
[I]: * Task B: priority=5 (as expected), wait_reason=DQUE_WRECEIVE (as expected), last_retval=
       TN_RC_DELETED (as expected)
[I]: * Task C: priority=4 (as expected), wait_reason=DQUE_WRECEIVE (as expected), last_retval=
       TN_RC_DELETED (as expected)
[I]: * Mutex M1: holder=NONE (as expected), lock_cnt=0 (as expected), exists=no (as expected)
```

If something goes wrong, there would be no `"as expected"`, but error and explanation what we expected and what we have. Tests halted.

I do my best to model nearly all possible situations within the each single subsystem (such as mutexes, queues, etc), including various situations with suspended tasks, deleted tasks, deleted objects, and the like. It helps a lot to keep the kernel really stable.

## 8.2 Get unit-tests

Currently, there is a separate repository with unit tests for TNeoKernel.

Please note that code of unit tests project is not as polished as the code of the kernel itself. My open-source time is limited, and I prefer to invest it in the kernel as much as possible.

Nevertheless, unit tests do their job efficiently, which is needed.

There is an "environment" repository, which contains tests and all the necessary library subrepos: http://hg.↩
dfrank.ru/tntest/_env
You can clone it as follows:

```
hg clone http://hg.dfrank.ru/tntest/_env tntest
```

The single repository with the tests resides here: http://hg.dfrank.ru/tntest/project_common

# Chapter 9

# Plans

I have plans to implement some goodies not yet present in the kernel.

## 9.1   Timer object

It would be sometimes useful to have ability to ask the kernel to call custom function in the future, or even call it periodically. This is what Timer object is for. The overhead of it would be the same as if we put some task to wait for specified timeout, i.e. it's really little overhead.

If we have no registered Timer objects, no extra overhead involved at all.

## 9.2   Event connecting

Sometimes we need to wait, say, for messages from several queues simultaneously; currently, the kernel does not have built-in support of it. How I plan to implement it: there should be a way to *connect* an event group and custom events mask to the data queue. Then, data queue will maintain flag(s) specified by mask: when queue is non-empty, it will set flag(s) by mask, when queue becomes empty, it will clear these flag(s).

Then, we can connect single event group to several queues passing different flags, and wait for the messages from all of these queues with just a single call to `tn_eventgrp_wait()` and friends. When event happened, we just check which flags are set, and get message from appropriate queue.

Probably it makes sense to connect event group to other objects as well, I haven't thought about this feature in detail yet. But, queue support is definitely must have.

# Chapter 10

# Legend

In the functions API documentation, the following designations are used:

-  Function can be called from task

-  Function can be called from ISR

-  Function can switch context to different task

-  Function can sleep

# Chapter 11

# Data Structure Index

## 11.1 Data Structures

Here are the data structures with brief descriptions:

# Chapter 12

# File Index

## 12.1 File List

Here is a list of all documented files with brief descriptions:

# Chapter 13

# Data Structure Documentation

## 13.1 TN_DQueue Struct Reference

### 13.1.1 Detailed Description

Structure representing data queue object.

Definition at line 83 of file tn_dqueue.h.

**Data Fields**

- struct TN_ListItem wait_send_list

    *list of tasks waiting to send data*
- struct TN_ListItem wait_receive_list

    *list of tasks waiting to receive data*
- void ** data_fifo

    *array of* `void *` *to store data queue items. Can be* `NULL`*.*
- int items_cnt

    *capacity (total items count). Can be 0.*
- int filled_items_cnt

    *count of non-free items in* `data_fifo`
- int head_idx

    *index of the item which will be written next time*
- int tail_idx

    *index of the item which will be read next time*
- enum TN_ObjId id_dque

    *id for object validity verification*

The documentation for this struct was generated from the following file:

- core/tn_dqueue.h

## 13.2 TN_DQueueTaskWait Struct Reference

### 13.2.1 Detailed Description

DQueue-specific fields related to waiting task, to be included in struct TN_Task.

Definition at line 115 of file tn_dqueue.h.

**Data Fields**

- void ∗ data_elem

    *if task tries to send the data to the data queue, and there's no space in the queue, value to put to queue is stored in this field*

The documentation for this struct was generated from the following file:

- core/tn_dqueue.h

## 13.3 TN_EGrpTaskWait Struct Reference

### 13.3.1 Detailed Description

EventGrp-specific fields related to waiting task, to be included in struct TN_Task.

Definition at line 123 of file tn_eventgrp.h.

**Data Fields**

- int wait_pattern

    *event wait pattern*
- enum TN_EGrpWaitMode wait_mode

    *event wait mode: AND or OR*
- int actual_pattern

    *pattern that caused task to finish waiting*

The documentation for this struct was generated from the following file:

- core/tn_eventgrp.h

## 13.4 TN_EventGrp Struct Reference

### 13.4.1 Detailed Description

Event group.

Definition at line 113 of file tn_eventgrp.h.

**Data Fields**

- struct TN_ListItem wait_queue

    *task wait queue*
- unsigned int pattern

    *current flags pattern*
- enum TN_ObjId id_event

    *id for object validity verification*

The documentation for this struct was generated from the following file:

- core/tn_eventgrp.h

## 13.5 TN_FMem Struct Reference

### 13.5.1 Detailed Description

Fixed memory blocks pool.

Definition at line 73 of file tn_fmem.h.

**Data Fields**

- struct TN_ListItem wait_queue

  *list of tasks waiting for free memory block*
- unsigned int block_size

  *block size (in bytes); note that it should be a multiple of* `sizeof(TN_UWord})`, *use a macro* `TN_MAKE_ALIG←_SIZE()` *for that.*
- int blocks_cnt

  *capacity (total blocks count)*
- int free_blocks_cnt

  *free blocks count*
- void ∗ start_addr

  *memory pool start address; note that it should be a multiple of* `sizeof(TN_UWord)`.
- void ∗ free_list

  *ptr to free block list*
- enum TN_ObjId id_fmp

  *id for object validity verification*

### 13.5.2 Field Documentation

#### 13.5.2.1 unsigned int TN_FMem::block_size

block size (in bytes); note that it should be a multiple of `sizeof(TN_UWord})`, use a macro `TN_MAKE_ALI←G_SIZE()` for that.

**See also**

    `TN_MAKE_ALIG_SIZE()`

Definition at line 83 of file tn_fmem.h.

#### 13.5.2.2 void∗ TN_FMem::start_addr

memory pool start address; note that it should be a multiple of `sizeof(TN_UWord)`.

Definition at line 93 of file tn_fmem.h.

The documentation for this struct was generated from the following file:

- core/tn_fmem.h

## 13.6 TN_FMemTaskWait Struct Reference

### 13.6.1 Detailed Description

FMem-specific fields related to waiting task, to be included in struct TN_Task.

Definition at line 107 of file tn_fmem.h.

**Data Fields**

- void ∗ data_elem

  *if task tries to receive data from memory pool, and there's no more free blocks in the pool, location to store pointer is saved in this field*

The documentation for this struct was generated from the following file:

- core/tn_fmem.h

## 13.7 TN_Mutex Struct Reference

### 13.7.1 Detailed Description

Mutex.

Definition at line 122 of file tn_mutex.h.

**Data Fields**

- struct TN_ListItem wait_queue

  *List of tasks that wait a mutex.*

- struct TN_ListItem mutex_queue

  *To include in task's locked mutexes list (if any)*

- struct TN_ListItem deadlock_list

  *List of other mutexes involved in deadlock (normally, this list is empty)*

- enum TN_MutexProtocol protocol

  *Mutex protocol: priority ceiling or priority inheritance.*

- struct TN_Task ∗ holder

  *Current mutex owner (task that locked mutex)*

- int ceil_priority

  *Used if only protocol is TN_MUTEX_PROT_CEILING: maximum priority of task that may lock the mutex.*

- int cnt

  *Lock count (for recursive locking)*

- enum TN_ObjId id_mutex

  *id for object validity verification*

The documentation for this struct was generated from the following file:

- core/tn_mutex.h

## 13.8 TN_Sem Struct Reference

### 13.8.1 Detailed Description

Semaphore.

Definition at line 88 of file tn_sem.h.

**Data Fields**

- struct TN_ListItem wait_queue

    *List of tasks that wait to acquire a semaphore.*
- int count

    *Current semaphore counter value.*
- int max_count

    *Max value of* `count`
- enum TN_ObjId id_sem

    *id for object validity verification*

The documentation for this struct was generated from the following file:

- core/tn_sem.h

## 13.9 TN_Task Struct Reference

### 13.9.1 Detailed Description

Task.

Definition at line 172 of file tn_tasks.h.

**Data Fields**

- unsigned int ∗ task_stk

    *pointer to task's current stack pointer; Note that this field* **must** *be a first field in the struct, this fact is exploited by platform-specific routines.*
- struct TN_ListItem task_queue

    *queue is used to include task in ready/wait lists*
- struct TN_ListItem timer_queue

    *queue is used to include task in timer list*
- struct TN_ListItem ∗ pwait_queue

    *pointer to object's (semaphore, mutex, event, etc) wait list in which task is included for waiting*
- struct TN_ListItem create_queue

    *queue is used to include task in creation list (currently, this list is used for statistics only)*
- struct TN_ListItem mutex_queue

    *list of all mutexes that are locked by task*
- struct TN_ListItem deadlock_list

    *list of other tasks involved in deadlock.*
- TN_UWord ∗ stk_start

    *base address of task's stack space*
- int stk_size

    *size of task's stack (in* `sizeof(unsigned int),` *not bytes)*
- TN_TaskBody ∗ task_func_addr

    *pointer to task's body function given to* `tn_task_create()`
- void ∗ task_func_param

    *pointer to task's parameter given to* `tn_task_create()`
- int base_priority

    *base priority of the task (actual current priority may be higher than base priority because of mutex)*
- int priority

      *current task priority*
- enum TN_ObjId id_task

      *id for object validity verification*
- enum TN_TaskState task_state

      *task state*
- enum TN_WaitReason task_wait_reason

      *reason for waiting (relevant if only* `task_state` *is* *WAIT* *or* *WAIT+SUSPEND*)
- enum TN_RCode task_wait_rc

      *waiting result code (reason why waiting finished)*
- unsigned long tick_count

      *remaining time until timeout; may be* *TN_WAIT_INFINITE*.
- int tslice_count

      *time slice counter*
- union {
    struct TN_EGrpTaskWait eventgrp
      *fields specific to tn_eventgrp.h*
    struct TN_DQueueTaskWait dqueue
      *fields specific to tn_dqueue.h*
    struct TN_FMemTaskWait fmem
      *fields specific to tn_fmem.h*
  } subsys_wait

      *subsystem-specific fields that are used while task waits for something.*
- const char ∗ name

      *task name for debug purposes, user may want to set it by hand*
- unsigned priority_already_updated: 1

      *Internal flag used to optimize mutex priority algorithms.*

## 13.9.2 Field Documentation

### 13.9.2.1 unsigned int∗ TN_Task::task_stk

pointer to task's current stack pointer; Note that this field **must** be a first field in the struct, this fact is exploited by platform-specific routines.

Definition at line 176 of file tn_tasks.h.

### 13.9.2.2 struct TN_ListItem TN_Task::deadlock_list

list of other tasks involved in deadlock.

This list is non-empty only in emergency cases, and it is here to help you fix your bug that led to deadlock.

**See also**

    TN_MUTEX_DEADLOCK_DETECT

Definition at line 203 of file tn_tasks.h.

### 13.9.2.3 union { ... } TN_Task::subsys_wait

subsystem-specific fields that are used while task waits for something.

Do note that these fields are grouped by union, so, they must not interfere with each other. It's quite ok here because task can't wait for different things.

**13.9.2.4    unsigned TN_Task::priority_already_updated**

Internal flag used to optimize mutex priority algorithms.

For the comments on it, see file tn_mutex.c, function `_mutex_do_unlock()`.

Definition at line 268 of file tn_tasks.h.

The documentation for this struct was generated from the following file:

- core/tn_tasks.h

# Chapter 14

# File Documentation

## 14.1 arch/example/tn_arch_example.h File Reference

### 14.1.1 Detailed Description

Example of architecture-dependent routines.

Definition in file tn_arch_example.h.

**Macros**

- #define _TN_FFS(x) (32 - __builtin_clz((x) & (0 - (x))))

  *FFS - find first set bit.*
- #define _TN_FATAL_ERROR(error_msg,...) {__asm__ volatile(" sdbbp 0"); __asm__ volatile ("nop");}

  *Used by the kernel as a signal that something really bad happened.*
- #define TN_ARCH_STK_ATTR_BEFORE

  *Compiler-specific attribute that should be placed **before** declaration of array used for stack.*
- #define TN_ARCH_STK_ATTR_AFTER __attribute__((aligned(0x8)))

  *Compiler-specific attribute that should be placed **after** declaration of array used for stack.*
- #define TN_MIN_STACK_SIZE 36

  *Minimum task's stack size, in words, not in bytes; includes a space for context plus for parameters passed to task's body function.*
- #define TN_INT_WIDTH 32

  *Width of `int` type.*
- #define TN_PRIORITIES_CNT TN_INT_WIDTH

  *Number of priorities available, this value usually matches `TN_INT_WIDTH`.*
- #define TN_WAIT_INFINITE 0xFFFFFFFF

  *Value for infinite waiting, usually matches `UINT_MAX`*
- #define TN_FILL_STACK_VAL 0xFEEDFACE

  *Value for initializing the task's stack.*
- #define TN_INTSAVE_DATA int tn_save_status_reg = 0;

  *Declares variable that is used by macros `TN_INT_DIS_SAVE()` and `TN_INT_RESTORE()` for storing status register value.*
- #define TN_INTSAVE_DATA_INT TN_INTSAVE_DATA

  *The same as `TN_INTSAVE_DATA` but for using in ISR together with `TN_INT_IDIS_SAVE()`, `TN_INT_IRE↩ STORE()`.*
- #define TN_INT_DIS_SAVE() tn_save_status_reg = tn_arch_sr_save_int_dis()

  *Disable interrupts and return previous value of status register, atomically.*

- #define TN_INT_RESTORE() tn_arch_sr_restore(tn_save_status_reg)

    *Restore previously saved status register.*
- #define TN_INT_IDIS_SAVE() TN_INT_DIS_SAVE()

    *The same as* `TN_INT_DIS_SAVE()` *but for using in ISR.*
- #define TN_INT_IRESTORE() TN_INT_RESTORE()

    *The same as* `TN_INT_RESTORE()` *but for using in ISR.*
- #define TN_IS_INT_DISABLED() ((__builtin_mfc0(12, 0) & 1) == 0)

    *Returns nonzero if interrupts are disabled, zero otherwise.*

## Typedefs

- typedef unsigned int TN_UWord

    *Unsigned integer type whose size is equal to the size of CPU register.*

### 14.1.2 Macro Definition Documentation

#### 14.1.2.1 #define _TN_FFS( *x* ) (32 - __builtin_clz((x) & (0 - (x))))

FFS - find first set bit.

Used in `_find_next_task_to_run()` function.

May be not defined: in this case, naive algorithm will be used.

Definition at line 52 of file tn_arch_example.h.

#### 14.1.2.2 #define _TN_FATAL_ERROR( *error_msg,* *...* ) {__asm__ volatile(" sdbbp 0"); __asm__ volatile ("nop");}

Used by the kernel as a signal that something really bad happened.

Indicates TNeoKernel bugs as well as illegal kernel usage (e.g. sleeping in the idle task callback)

Typically, set to assembler instruction that causes debugger to halt.

Definition at line 61 of file tn_arch_example.h.

#### 14.1.2.3 #define TN_ARCH_STK_ATTR_BEFORE

Compiler-specific attribute that should be placed **before** declaration of array used for stack.

It is needed because there are often additional restrictions applied to alignment of stack, so, to meet them, stack arrays need to be declared with these macros.

**See also**

    TN_ARCH_STK_ATTR_AFTER

Definition at line 75 of file tn_arch_example.h.

#### 14.1.2.4 #define TN_ARCH_STK_ATTR_AFTER __attribute__((aligned(0x8)))

Compiler-specific attribute that should be placed **after** declaration of array used for stack.

It is needed because there are often additional restrictions applied to alignment of stack, so, to meet them, stack arrays need to be declared with these macros.

**See also**

> TN_ARCH_STK_ATTR_BEFORE

Definition at line 86 of file tn_arch_example.h.

### 14.1.2.5 #define TN_PRIORITIES_CNT TN_INT_WIDTH

Number of priorities available, this value usually matches TN_INT_WIDTH.

For compatibility with all platforms, it's recommended to use only values from 1 to 14, inclusive.

**See also**

> TN_INT_WIDTH

Definition at line 112 of file tn_arch_example.h.

### 14.1.2.6 #define TN_INTSAVE_DATA int tn_save_status_reg = 0;

Declares variable that is used by macros TN_INT_DIS_SAVE() and TN_INT_RESTORE() for storing status register value.

**See also**

> TN_INT_DIS_SAVE()
> TN_INT_RESTORE()

Definition at line 134 of file tn_arch_example.h.

### 14.1.2.7 #define TN_INTSAVE_DATA_INT TN_INTSAVE_DATA

The same as TN_INTSAVE_DATA but for using in ISR together with TN_INT_IDIS_SAVE(), TN_INT_IR↩
ESTORE().

**See also**

> TN_INT_IDIS_SAVE()
> TN_INT_IRESTORE()

Definition at line 143 of file tn_arch_example.h.

### 14.1.2.8 #define TN_INT_DIS_SAVE(  ) tn_save_status_reg = tn_arch_sr_save_int_dis()

Disable interrupts and return previous value of status register, atomically.

Similar tn_arch_sr_save_int_dis(), but implemented as a macro, so it is potentially faster.

Uses TN_INTSAVE_DATA as a temporary storage.

**See also**

> TN_INTSAVE_DATA
> tn_arch_sr_save_int_dis()

Definition at line 155 of file tn_arch_example.h.

**14.1.2.9   #define TN_INT_RESTORE(   ) tn_arch_sr_restore(tn_save_status_reg)**

Restore previously saved status register.

Similar to tn_arch_sr_restore(), but implemented as a macro, so it is potentially faster.

Uses TN_INTSAVE_DATA as a temporary storage.

**See also**

>    TN_INTSAVE_DATA
>    tn_arch_sr_save_int_dis()

Definition at line 167 of file tn_arch_example.h.

**14.1.2.10   #define TN_INT_IDIS_SAVE(   ) TN_INT_DIS_SAVE()**

The same as TN_INT_DIS_SAVE() but for using in ISR.

Uses TN_INTSAVE_DATA_INT as a temporary storage.

**See also**

>    TN_INTSAVE_DATA_INT

Definition at line 176 of file tn_arch_example.h.

**14.1.2.11   #define TN_INT_IRESTORE(   ) TN_INT_RESTORE()**

The same as TN_INT_RESTORE() but for using in ISR.

Uses TN_INTSAVE_DATA_INT as a temporary storage.

**See also**

>    TN_INTSAVE_DATA_INT

Definition at line 185 of file tn_arch_example.h.

## 14.1.3   Typedef Documentation

**14.1.3.1   typedef unsigned int TN_UWord**

Unsigned integer type whose size is equal to the size of CPU register.

Typically it's plain `unsigned int.`

Definition at line 103 of file tn_arch_example.h.

## 14.2   arch/pic32/tn_arch_pic32.h File Reference

### 14.2.1   Detailed Description

PIC32 architecture-dependent routines.

Definition in file tn_arch_pic32.h.

**Macros**

- #define tn_soft_isr(vec)

    *Interrupt handler wrapper macro for software context saving.*

- #define tn_srs_isr(vec)

    *Interrupt handler wrapper macro for shadow register context saving.*

## 14.2.2 Macro Definition Documentation

### 14.2.2.1 #define tn_soft_isr( *vec* )

Interrupt handler wrapper macro for software context saving.

Usage looks like the following:

```
tn_soft_isr(_TIMER_1_VECTOR)
{
   INTClearFlag(INT_T1);

   //-- do something useful
}
```

Note that you should not use `__ISR(_TIMER_1_VECTOR)` macro for that.

**Parameters**

| | |
|---:|---|
| *vec* | interrupt vector number, such as `_TIMER_1_VECTOR`, etc. |

Definition at line 269 of file tn_arch_pic32.h.

### 14.2.2.2 #define tn_srs_isr( *vec* )

Interrupt handler wrapper macro for shadow register context saving.

Usage looks like the following:

```
tn_srs_isr(_INT_UART_1_VECTOR)
{
   INTClearFlag(INT_U1);

   //-- do something useful
}
```

Note that you should not use `__ISR(_INT_UART_1_VECTOR)` macro for that.

**Parameters**

| | |
|---:|---|
| *vec* | interrupt vector number, such as `_TIMER_1_VECTOR`, etc. |

Definition at line 429 of file tn_arch_pic32.h.

## 14.3 arch/tn_arch.h File Reference

### 14.3.1 Detailed Description

Architecture-dependent routines declaration.

Definition in file tn_arch.h.

**Functions**

- void tn_arch_int_dis (void)

  *Unconditionally disable interrupts.*
- void tn_arch_int_en (void)

  *Unconditionally enable interrupts.*
- unsigned int tn_arch_sr_save_int_dis (void)

  *Disable interrupts and return previous value of status register, atomically.*
- void tn_arch_sr_restore (unsigned int sr)

  *Restore previously saved status register.*
- TN_UWord ∗ _tn_arch_stack_start_get (TN_UWord ∗stack_low_address, int stack_size)

  *Should return start stack address, which may be either the lowest address of the stack array or the highest one, depending on the architecture.*
- unsigned int ∗ _tn_arch_stack_init (TN_TaskBody ∗task_func, TN_UWord ∗stack_start, void ∗param)

  *Should initialize stack for new task and return current stack pointer.*
- int _tn_arch_inside_isr (void)

  *Should return 1 if ISR is currently running, 0 otherwise.*
- void _tn_arch_context_switch (void)

  *Called whenever we need to switch context to other task.*
- void _tn_arch_context_switch_exit (void)

  *Called when some task calls* `tn_task_exit()`.
- void _tn_arch_system_start (void)

  *Should perform first context switch (to the task pointed to by* `tn_next_task_to_run`).

### 14.3.2 Function Documentation

#### 14.3.2.1 unsigned int tn_arch_sr_save_int_dis ( void )

Disable interrupts and return previous value of status register, atomically.

**See also**

```
tn_arch_sr_restore()
```

#### 14.3.2.2 void tn_arch_sr_restore ( unsigned int *sr* )

Restore previously saved status register.

**Parameters**

| | |
|---|---|
| *sr* | status register value previously from `tn_arch_sr_save_int_dis()` |

**See also**

```
tn_arch_sr_save_int_dis()
```

#### 14.3.2.3 TN_UWord∗ _tn_arch_stack_start_get ( TN_UWord ∗ *stack_low_address,* int *stack_size* )

Should return start stack address, which may be either the lowest address of the stack array or the highest one, depending on the architecture.

**Parameters**

| | |
|---:|---|
| *stack_low_↩* *address* | start address of the stack array. |
| *stack_size* | size of the stack in `int`-s, not in bytes. |

**14.3.2.4  unsigned int∗ _tn_arch_stack_init ( TN_TaskBody ∗ *task_func,* TN_UWord ∗ *stack_start,* void ∗ *param* )**

Should initialize stack for new task and return current stack pointer.

**Parameters**

| | |
|---:|---|
| *task_func* | Pointer to task body function. |
| *stack_start* | Start address of the stack, returned by `_tn_arch_stack_start_get()`. |
| *param* | User-provided parameter for task body function. |

**Returns**

current stack pointer (top of the stack)

**14.3.2.5  void _tn_arch_context_switch ( void )**

Called whenever we need to switch context to other task.

**Preconditions:**

- interrupts are enabled;

- `tn_curr_run_task` points to currently running (preempted) task;

- `tn_next_task_to_run` points to new task to run.

**Actions to perform:**

- save context of the preempted task to its stack;

- set `tn_curr_run_task` to `tn_next_task_to_run`;

- switch context to it.

**See also**

```
tn_curr_run_task
tn_next_task_to_run
```

**14.3.2.6  void _tn_arch_context_switch_exit ( void )**

Called when some task calls `tn_task_exit()`.

**Preconditions:**

- interrupts are disabled;

- `tn_next_task_to_run` is already set to other task.

**Actions to perform:**

- set `tn_curr_run_task` to `tn_next_task_to_run`;

- switch context to it.

**See also**

```
tn_curr_run_task
tn_next_task_to_run
```

**14.3.2.7   void _tn_arch_system_start ( void )**

Should perform first context switch (to the task pointed to by `tn_next_task_to_run`).

**Preconditions:**

- no interrupts are set up yet, so, it's like interrupts disabled

- `tn_next_task_to_run` is already set to idle task.

**Actions to perform:**

- set `TN_STATE_FLAG__SYS_RUNNING` flag in the `tn_sys_state` variable;

- set `tn_curr_run_task` to `tn_next_task_to_run`;

- switch context to it.

**See also**

```
TN_STATE_FLAG__SYS_RUNNING
tn_sys_state
tn_curr_run_task
tn_next_task_to_run
```

## 14.4   core/tn_common.h File Reference

### 14.4.1   Detailed Description

Definitions used through the whole kernel.

Definition in file tn_common.h.

**Macros**

- #define TN_API_MAKE_ALIG_ARG__TYPE 1

    *In this case, you should use macro like this:* `TN_MAKE_ALIG(struct my_struct).`
- #define TN_API_MAKE_ALIG_ARG__SIZE 2

    *In this case, you should use macro like this:* `TN_MAKE_ALIG(sizeof(struct my_struct)).`
- #define NULL ((void ∗)0)

    *NULL pointer definition.*
- #define BOOL int

    *boolean type definition*
- #define TRUE (1 == 1)

    `true` *value definition for type* `BOOL`
- #define FALSE (1 == 0)

    `false` *value definition for type* `BOOL`
- #define TN_MAKE_ALIG_SIZE(a) (((a) + (sizeof(TN_UWord) - 1)) & (∼(sizeof(TN_UWord) - 1)))

    *Macro for making a number a multiple of* `sizeof(TN_UWord)`, *should be used with fixed memory block pool.*
- #define TN_MAKE_ALIG(a) TN_MAKE_ALIG_SIZE(a)

    *The same as* `TN_MAKE_ALIG_SIZE` *but its behavior depends on the option* `TN_API_MAKE_ALIG_ARG`

**Typedefs**

- typedef void( TN_TaskBody )(void ∗param)

    *Prototype for task body function.*
- typedef unsigned long TN_Timeout

    *The value representing maximum number of system ticks to wait.*

**Enumerations**

- enum TN_ObjId {
  TN_ID_TASK = 0x47ABCF69, TN_ID_SEMAPHORE = 0x6FA173EB, TN_ID_EVENTGRP = 0x5E224F25,
  TN_ID_DATAQUEUE = 0x8C8A6C89,
  TN_ID_FSMEMORYPOOL = 0x26B7CE8B, TN_ID_MUTEX = 0x17129E45 }

    *Magic number for object validity verification.*
- enum TN_RCode {
  TN_RC_OK = 0, TN_RC_TIMEOUT = -1, TN_RC_OVERFLOW = -2, TN_RC_WCONTEXT = -3,
  TN_RC_WSTATE = -4, TN_RC_WPARAM = -5, TN_RC_ILLEGAL_USE = -6, TN_RC_INVALID_OBJ = -7,
  TN_RC_DELETED = -8, TN_RC_FORCED = -9, TN_RC_INTERNAL = -10 }

    *Result code returned by kernel services.*

### 14.4.2 Macro Definition Documentation

#### 14.4.2.1 #define TN_API_MAKE_ALIG_ARG__TYPE 1

In this case, you should use macro like this: `TN_MAKE_ALIG(struct my_struct)`.

This way is used in the majority of TNKernel ports. (actually, in all ports except the one by AlexB)

Definition at line 56 of file tn_common.h.

#### 14.4.2.2 #define TN_API_MAKE_ALIG_ARG__SIZE 2

In this case, you should use macro like this: `TN_MAKE_ALIG(sizeof(struct my_struct))`.

This way is stated in TNKernel docs and used in the port for dsPIC/PIC24/PIC32 by AlexB.

Definition at line 63 of file tn_common.h.

#### 14.4.2.3 #define TN_MAKE_ALIG_SIZE( *a* ) (((a) + (sizeof(TN_UWord) - 1)) & (∼(sizeof(TN_UWord) - 1)))

Macro for making a number a multiple of `sizeof(TN_UWord)`, should be used with fixed memory block pool.

See `tn_fmem_create()` for usage example.

Definition at line 239 of file tn_common.h.

#### 14.4.2.4 #define TN_MAKE_ALIG( *a* ) TN_MAKE_ALIG_SIZE(a)

The same as `TN_MAKE_ALIG_SIZE` but its behavior depends on the option `TN_API_MAKE_ALIG_ARG`

**Attention**

it is recommended to use `TN_MAKE_ALIG_SIZE` macro instead of this one, in order to avoid confusion caused by various TNKernel ports: refer to the section Macro MAKE_ALIG() for details.

Definition at line 263 of file tn_common.h.

### 14.4.3 Typedef Documentation

#### 14.4.3.1 typedef unsigned long **TN_Timeout**

The value representing maximum number of system ticks to wait.

Assume user called some system function, and it can't perform its job immediately (say, it needs to lock mutex but it is already locked, etc).

So, function can wait or return an error. There are possible `timeout` values and appropriate behavior of the function:

- `timeout` is set to `0`: function doesn't wait at all, no context switch is performed, `TN_RC_TIMEOUT` is returned immediately.

- `timeout` is set to `TN_WAIT_INFINITE`: function waits until it eventually **can** perform its job. Timeout is not taken in account, so `TN_RC_TIMEOUT` is never returned.

- `timeout` is set to other value: function waits at most specified number of system ticks. Strictly speaking, it waits from (`timeout - 1`) to `timeout` ticks. So, if you specify that timeout is 1, be aware that it might actually don't wait at all: if system timer interrupt happens just while function is putting task to wait (with interrupts disabled), then ISR will be executed right after function puts task to wait. Then `tn_tick_int↵ _processing()` will immediately remove the task from wait queue and make it runnable again.

    So, to guarantee that task waits *at least* 1 system tick, you should specify timeout value of `2`.

**Note** also that there are other possible ways to make task runnable:

- if task waits because of call to `tn_task_sleep()`, it may be woken up by some other task, by means of `tn_task_wakeup()`. In this case, `tn_task_sleep()` returns `TN_RC_OK`.

- independently of the wait reason, task may be released from wait forcibly, by means of `tn_task_↵ release_wait()`. It this case, `TN_RC_FORCED` is returned by the waiting function. (the usage of the `tn_task_release_wait()` function is discouraged though)

Definition at line 203 of file tn_common.h.

### 14.4.4 Enumeration Type Documentation

#### 14.4.4.1 enum **TN_ObjId**

Magic number for object validity verification.

**Enumerator**

**TN_ID_TASK**  id for tasks

**TN_ID_SEMAPHORE**  id for semaphores

**TN_ID_EVENTGRP**  id for event groups

**TN_ID_DATAQUEUE**  id for data queues

**TN_ID_FSMEMORYPOOL**  id for fixed memory pools

**TN_ID_MUTEX**  id for mutexes

Definition at line 87 of file tn_common.h.

**14.4.4.2 enum TN_RCode**

Result code returned by kernel services.

**Enumerator**

**TN_RC_OK** Successful operation.

**TN_RC_TIMEOUT** Timeout (consult `TN_Timeout` for details).

> **See also**
>
> > `TN_Timeout`

**TN_RC_OVERFLOW** This code is returned in the following cases:

- Trying to increment semaphore count more than its max count;
- Trying to return extra memory block to fixed memory pool.

  > **See also**
  >
  > > `tn_sem.h`
  > > `tn_fmem.h`

**TN_RC_WCONTEXT** Wrong context error: returned if function is called from non-acceptable context. Required context suggested for every function by badges:

-  - function can be called from task;

-  - function can be called from ISR.

> **See also**
>
> > `tn_sys_context_get()`
> > enum `TN_Context`

**TN_RC_WSTATE** Wrong task state error: requested operation requires different task state.

**TN_RC_WPARAM** This code is returned by most of the kernel functions when wrong params were given to function. This error code can be returned if only build-time option `TN_CHECK_PARAM` is non-zero

> **See also**
>
> > `TN_CHECK_PARAM`

**TN_RC_ILLEGAL_USE** Illegal usage. Returned in the following cases:

- task tries to unlock or delete the mutex that is locked by different task,
- task tries to lock mutex with priority ceiling whose priority is lower than task's priority

  > **See also**
  >
  > > `tn_mutex.h`

**TN_RC_INVALID_OBJ** Returned when user tries to perform some operation on invalid object (mutex, semaphore, etc). Object validity is checked by comparing special `id_...` field value with the value from enum `TN_ObjId`

> **See also**
>
> > `TN_CHECK_PARAM`

**TN_RC_DELETED** Object for whose event task was waiting is deleted.

**TN_RC_FORCED** Task was released from waiting forcibly because some other task called `tn_task_↩ release_wait()`

**TN_RC_INTERNAL** Internal kernel error, should never be returned by kernel services. If it is returned, it's a bug in the kernel.

Definition at line 99 of file tn_common.h.

## 14.5  core/tn_dqueue.h File Reference

### 14.5.1  Detailed Description

A data queue is a FIFO that stores pointer (of type `void *`) in each cell, called (in uITRON 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 tries 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`).

Definition in file tn_dqueue.h.

### Data Structures

- struct TN_DQueue

    *Structure representing data queue object.*
- struct TN_DQueueTaskWait

    *DQueue-specific fields related to waiting task, to be included in struct TN_Task.*

### Functions

- enum TN_RCode tn_queue_create (struct TN_DQueue *dque, void **data_fifo, int items_cnt)

    *Construct data queue.*
- enum TN_RCode tn_queue_delete (struct TN_DQueue *dque)

    *Destruct data queue.*
- enum TN_RCode tn_queue_send (struct TN_DQueue *dque, void *p_data, TN_Timeout timeout)

    *Send the data element specified by the `p_data` to the data queue specified by the `dque`.*
- enum TN_RCode tn_queue_send_polling (struct TN_DQueue *dque, void *p_data)

    *The same as `tn_queue_send()` with zero timeout.*
- enum TN_RCode tn_queue_isend_polling (struct TN_DQueue *dque, void *p_data)

    *The same as `tn_queue_send()` with zero timeout, but for using in the ISR.*
- enum TN_RCode tn_queue_receive (struct TN_DQueue *dque, void **pp_data, TN_Timeout timeout)

    *Receive the data element from the data queue specified by the `dque` and place it into the address specified by the `pp_data`.*
- enum TN_RCode tn_queue_receive_polling (struct TN_DQueue *dque, void **pp_data)

    *The same as `tn_queue_receive()` with zero timeout.*
- enum TN_RCode tn_queue_ireceive_polling (struct TN_DQueue *dque, void **pp_data)

    *The same as `tn_queue_receive()` with zero timeout, but for using in the ISR.*

### 14.5.2  Function Documentation

#### 14.5.2.1  enum **TN_RCode** tn_queue_create (  struct **TN_DQueue** * *dque,*  void ** *data_fifo,*  int *items_cnt* )

Construct data queue.

`id_dque` member should not contain `TN_ID_DATAQUEUE`, otherwise, `TN_RC_WPARAM` is returned.

*(refer to Legend for details)*

**Parameters**

| | |
|---|---|
| *dque* | pointer to already allocated struct TN_DQueue. |
| *data_fifo* | pointer to already allocated array of `void *` to store data queue items. Can be NULL. |
| *items_cnt* | capacity of queue (count of elements in the `data_fifo` array) Can be 0. |

**Returns**

- TN_RC_OK if queue was successfully created;

- If TN_CHECK_PARAM is non-zero, additional return code is available: TN_RC_WPARAM.

**14.5.2.2    enum TN_RCode tn_queue_delete ( struct TN_DQueue ∗ *dque* )**

Destruct data queue.

All tasks that wait for writing to or reading from the queue become runnable with TN_RC_DELETED code returned. TN_RCode, struct TN_Task.

(refer to *Legend* for details)

**Parameters**

| | |
|---|---|
| *dque* | pointer to data queue to be deleted |

**Returns**

- TN_RC_OK if queue was successfully deleted;

- TN_RC_WCONTEXT if called from wrong context;

- If TN_CHECK_PARAM is non-zero, additional return codes are available: TN_RC_WPARAM and TN_←RC_INVALID_OBJ.

**14.5.2.3    enum TN_RCode tn_queue_send ( struct TN_DQueue ∗ *dque,* void ∗ *p_data,* TN_Timeout *timeout* )**

Send the data element specified by the `p_data` 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 `p_data` to task's function, that caused it to wait.

If there are no tasks in the data queue's `wait_receive` list, parameter `p_data` is placed to the tail of data FIFO. If the data FIFO is full, behavior depends on the `timeout` value: refer to TN_Timeout.

(refer to *Legend* for details)

**Parameters**

| | |
|---|---|
| *dque* | pointer to data queue to send data to |
| *p_data* | value to send |
| *timeout* | refer to TN_Timeout |

**Returns**

- TN_RC_OK if data was successfully sent;

- TN_RC_WCONTEXT if called from wrong context;

- Other possible return codes depend on `timeout` value, refer to TN_Timeout

- If TN_CHECK_PARAM is non-zero, additional return codes are available: TN_RC_WPARAM and TN_←RC_INVALID_OBJ.

**See also**

> `TN_Timeout`

**14.5.2.4    enum TN_RCode tn_queue_send_polling ( struct TN_DQueue ∗ *dque,* void ∗ *p_data* )**

The same as `tn_queue_send()` with zero timeout.

 *(refer to Legend for details)*

**14.5.2.5    enum TN_RCode tn_queue_isend_polling ( struct TN_DQueue ∗ *dque,* void ∗ *p_data* )**

The same as `tn_queue_send()` with zero timeout, but for using in the ISR.

 *(refer to Legend for details)*

**14.5.2.6    enum TN_RCode tn_queue_receive ( struct TN_DQueue ∗ *dque,* void ∗∗ *pp_data,* TN_Timeout *timeout* )**

Receive the data element from the data queue specified by the `dque` and place it into the address specified by the `pp_data`.

If the FIFO already has data, function removes an entry from the end of the data queue FIFO and returns it into the `pp_data` function parameter.

If there are task(s) in the data queue's `wait_send` list, first one gets removed from the head of `wait_send` list, becomes 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, behavior depends on the `timeout` value: refer to `TN_Timeout`.

 *(refer to Legend for details)*

**Parameters**

| | |
|---|---|
| *dque* | pointer to data queue to receive data from |
| *pp_data* | pointer to location to store the value |
| *timeout* | refer to `TN_Timeout` |

**Returns**

- `TN_RC_OK` if data was successfully received;

- `TN_RC_WCONTEXT` if called from wrong context;

- Other possible return codes depend on `timeout` value, refer to `TN_Timeout`

- If `TN_CHECK_PARAM` is non-zero, additional return codes are available: `TN_RC_WPARAM` and `TN_↩RC_INVALID_OBJ`.

**See also**

> `TN_Timeout`

**14.5.2.7    enum TN_RCode tn_queue_receive_polling ( struct TN_DQueue ∗ *dque,* void ∗∗ *pp_data* )**

The same as `tn_queue_receive()` with zero timeout.

 *(refer to Legend for details)*

**14.5.2.8   enum TN_RCode tn_queue_ireceive_polling ( struct TN_DQueue** ∗ *dque,* **void** ∗∗ *pp_data* **)**

The same as `tn_queue_receive()` with zero timeout, but for using in the ISR.

  *(refer to Legend for details)*

## 14.6   core/tn_eventgrp.h File Reference

### 14.6.1   Detailed Description

Event group.

An event group has an internal variable (of type `unsigned int`), which is interpreted as a bit pattern where each bit represents an event. An event group 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.

The tasks waiting for an event(s) are placed in the event group's wait queue. An event group 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.

Definition in file tn_eventgrp.h.

**Data Structures**

- struct TN_EventGrp

    *Event group.*
- struct TN_EGrpTaskWait

    *EventGrp-specific fields related to waiting task, to be included in struct TN_Task.*

**Enumerations**

- enum TN_EGrpWaitMode { TN_EVENTGRP_WMODE_OR = (1 << 0), TN_EVENTGRP_WMODE_AND = (1 << 1) }

    *Events waiting mode: wait for all flags to be set or just for any of the specified flags to be set.*
- enum TN_EGrpOp { TN_EVENTGRP_OP_SET, TN_EVENTGRP_OP_CLEAR, TN_EVENTGRP_OP_TO↩GGLE }

    *Modify operation: set, clear or toggle.*

**Functions**

- enum TN_RCode tn_eventgrp_create (struct TN_EventGrp ∗eventgrp, unsigned int initial_pattern)

    *Construct event group.*
- enum TN_RCode tn_eventgrp_delete (struct TN_EventGrp ∗eventgrp)

    *Destruct event group.*
- enum TN_RCode tn_eventgrp_wait (struct TN_EventGrp ∗eventgrp, unsigned int wait_pattern, enum TN_↩EGrpWaitMode wait_mode, unsigned int ∗p_flags_pattern, TN_Timeout timeout)

    *Wait for specified event(s) in the event group.*
- enum TN_RCode tn_eventgrp_wait_polling (struct TN_EventGrp ∗eventgrp, unsigned int wait_pattern, enum TN_EGrpWaitMode wait_mode, unsigned int ∗p_flags_pattern)

    *The same as* `tn_eventgrp_wait()` *with zero timeout.*
- enum TN_RCode tn_eventgrp_iwait_polling (struct TN_EventGrp ∗eventgrp, unsigned int wait_pattern, enum TN_EGrpWaitMode wait_mode, unsigned int ∗p_flags_pattern)

    *The same as* `tn_eventgrp_wait()` *with zero timeout, but for using in the ISR.*

- enum TN_RCode tn_eventgrp_modify (struct TN_EventGrp ∗eventgrp, enum TN_EGrpOp operation, unsigned int pattern)

  *Modify current events bit pattern in the event group.*

- enum TN_RCode tn_eventgrp_imodify (struct TN_EventGrp ∗eventgrp, enum TN_EGrpOp operation, unsigned int pattern)

  *The same as* `tn_eventgrp_modify()`, *but for using in the ISR.*

## 14.6.2 Enumeration Type Documentation

### 14.6.2.1 enum **TN_EGrpWaitMode**

Events waiting mode: wait for all flags to be set or just for any of the specified flags to be set.

**Enumerator**

> **TN_EVENTGRP_WMODE_OR** Task waits for **any** of the event bits from the `wait_pattern` to be set in the event group.
>
> **TN_EVENTGRP_WMODE_AND** Task waits for **all** of the event bits from the `wait_pattern` to be set in the event group.

Definition at line 78 of file tn_eventgrp.h.

### 14.6.2.2 enum **TN_EGrpOp**

Modify operation: set, clear or toggle.

To be used in `tn_eventgrp_modify()` / `tn_eventgrp_imodify()` functions.

**Enumerator**

> **TN_EVENTGRP_OP_SET** Set flags that are set in given `pattern` argument. Note that this operation can lead to the context switch, since other high-priority task(s) might wait for the event.
>
> **TN_EVENTGRP_OP_CLEAR** Clear flags that are set in the given `pattern` argument. This operation can **not** lead to the context switch, since tasks can't wait for events to be cleared.
>
> **TN_EVENTGRP_OP_TOGGLE** Toggle flags that are set in the given `pattern` argument. Note that this operation can lead to the context switch, since other high-priority task(s) might wait for the event that was just set (if any).

Definition at line 93 of file tn_eventgrp.h.

## 14.6.3 Function Documentation

### 14.6.3.1 enum TN_RCode tn_eventgrp_create ( struct **TN_EventGrp** ∗ *eventgrp,* unsigned int *initial_pattern* )

Construct event group.

`id_event` field should not contain `TN_ID_EVENTGRP`, otherwise, `TN_RC_WPARAM` is returned.

 *(refer to Legend for details)*

**Parameters**

| eventgrp | Pointer to already allocated struct TN_EventGrp |
| initial_pattern | Initial events pattern. |

**Returns**

- TN_RC_OK if event group was successfully created;

- If TN_CHECK_PARAM is non-zero, additional return code is available: TN_RC_WPARAM.

**14.6.3.2 enum TN_RCode tn_eventgrp_delete ( struct TN_EventGrp ∗ eventgrp )**

Destruct event group.

All tasks that wait for the event(s) become runnable with TN_RC_DELETED code returned.

 *(refer to Legend for details)*

**Parameters**

| eventgrp | Pointer to event groupt to be deleted. |

**Returns**

- TN_RC_OK if event group was successfully deleted;

- TN_RC_WCONTEXT if called from wrong context;

- If TN_CHECK_PARAM is non-zero, additional return codes are available: TN_RC_WPARAM and TN_↩
  RC_INVALID_OBJ.

**14.6.3.3 enum TN_RCode tn_eventgrp_wait ( struct TN_EventGrp ∗ eventgrp, unsigned int wait_pattern, enum TN_EGrpWaitMode wait_mode, unsigned int ∗ p_flags_pattern, TN_Timeout timeout )**

Wait for specified event(s) in the event group.

If the specified event is already active, function returns TN_RC_OK immediately. Otherwise, behavior depends on timeout value: refer to TN_Timeout.

 *(refer to Legend for details)*

**Parameters**

| eventgrp | Pointer to event group to wait events from |
| wait_pattern | Events bit pattern for which task should wait |
| wait_mode | Specifies whether task should wait for **all** the event bits from wait_pattern to be set, or for just **any** of them (see enum TN_EGrpWaitMode) |
| p_flags_pattern | Pointer to the unsigned int variable in which actual event pattern that caused task to stop waiting will be stored. May be NULL. |
| timeout | refer to TN_Timeout |

**Returns**

- TN_RC_OK if specified event is active (so the task can check variable pointed to by p_flags_↩
  pattern if it wasn't NULL).

- TN_RC_WCONTEXT if called from wrong context;

- Other possible return codes depend on timeout value, refer to TN_Timeout

- If TN_CHECK_PARAM is non-zero, additional return codes are available: TN_RC_WPARAM and TN_↩
  RC_INVALID_OBJ.

**14.6.3.4   enum TN_RCode tn_eventgrp_wait_polling ( struct TN_EventGrp ∗ *eventgrp,* unsigned int *wait_pattern,* enum TN_EGrpWaitMode *wait_mode,* unsigned int ∗ *p_flags_pattern* )**

The same as `tn_eventgrp_wait()` with zero timeout.

*(refer to Legend for details)*


**14.6.3.5   enum TN_RCode tn_eventgrp_iwait_polling ( struct TN_EventGrp ∗ *eventgrp,* unsigned int *wait_pattern,* enum TN_EGrpWaitMode *wait_mode,* unsigned int ∗ *p_flags_pattern* )**

The same as `tn_eventgrp_wait()` with zero timeout, but for using in the ISR.

*(refer to Legend for details)*


**14.6.3.6   enum TN_RCode tn_eventgrp_modify ( struct TN_EventGrp ∗ *eventgrp,* enum TN_EGrpOp *operation,* unsigned int *pattern* )**

Modify current events bit pattern in the event group.

Behavior depends on the given `operation`: refer to enum `TN_EGrpOp`

*(refer to Legend for details)*

**Parameters**

| | |
|---|---|
| *eventgrp* | Pointer to event group to modify events in |
| *operation* | Actual operation to perform: set, clear or toggle. Refer to enum `TN_EGrpOp` |
| *pattern* | Events pattern to be applied (depending on `operation` value) |

**Returns**

- `TN_RC_OK` on success;
- `TN_RC_WCONTEXT` if called from wrong context;
- If `TN_CHECK_PARAM` is non-zero, additional return codes are available: `TN_RC_WPARAM` and `TN_↩RC_INVALID_OBJ`.


**14.6.3.7   enum TN_RCode tn_eventgrp_imodify ( struct TN_EventGrp ∗ *eventgrp,* enum TN_EGrpOp *operation,* unsigned int *pattern* )**

The same as `tn_eventgrp_modify()`, but for using in the ISR.

*(refer to Legend for details)*


## 14.7   core/tn_fmem.h File Reference

### 14.7.1   Detailed Description

Fixed memory blocks pool.

A fixed-sized memory blocks pool is used for managing fixed-sized memory blocks dynamically. A 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).

Definition in file tn_fmem.h.

**Data Structures**

- struct TN_FMem

    *Fixed memory blocks pool.*
- struct TN_FMemTaskWait

    *FMem-specific fields related to waiting task, to be included in struct TN_Task.*

**Macros**

- #define TN_FMEM_BUF_DEF(name, item_type, size)

    *Convenience macro for the definition of buffer for memory pool.*

**Functions**

- enum TN_RCode tn_fmem_create (struct TN_FMem ∗fmem, void ∗start_addr, unsigned int block_size, int blocks_cnt)

    *Construct fixed memory blocks pool.*
- enum TN_RCode tn_fmem_delete (struct TN_FMem ∗fmem)

    *Destruct fixed memory blocks pool.*
- enum TN_RCode tn_fmem_get (struct TN_FMem ∗fmem, void ∗∗p_data, TN_Timeout timeout)

    *Get memory block from the pool.*
- enum TN_RCode tn_fmem_get_polling (struct TN_FMem ∗fmem, void ∗∗p_data)

    *The same as* `tn_fmem_get()` *with zero timeout.*
- enum TN_RCode tn_fmem_iget_polling (struct TN_FMem ∗fmem, void ∗∗p_data)

    *The same as* `tn_fmem_get()` *with zero timeout, but for using in the ISR.*
- enum TN_RCode tn_fmem_release (struct TN_FMem ∗fmem, void ∗p_data)

    *Release memory block back to the pool.*
- enum TN_RCode tn_fmem_irelease (struct TN_FMem ∗fmem, void ∗p_data)

    *The same as* `tn_fmem_get()`, *but for using in the ISR.*

### 14.7.2 Macro Definition Documentation

#### 14.7.2.1 #define TN_FMEM_BUF_DEF( *name, item_type, size* )

**Value:**

```
TN_UWord name[                                             \
      (size)                                              \
    * (TN_MAKE_ALIG_SIZE(sizeof(item_type)) / sizeof(TN_UWord)) \
    ]
```

Convenience macro for the definition of buffer for memory pool.

See `tn_fmem_create()` for usage example.

**Parameters**

| | |
|---:|:---|
| *name* | C variable name of the buffer array (this name should be given to the `tn_fmem_create()` function as the `start_addr` argument) |
| *item_type* | Type of item in the memory pool, like `struct MyMemoryItem`. |

| | |
|---:|:---|
| *size* | Number of items in the memory pool. |

Definition at line 135 of file tn_fmem.h.

### 14.7.3 Function Documentation

#### 14.7.3.1 enum **TN_RCode** tn_fmem_create ( struct **TN_FMem** ∗ *fmem,* void ∗ *start_addr,* unsigned int *block_size,* int *blocks_cnt* )

Construct fixed memory blocks pool.

`id_fmp` field should not contain TN_ID_FSMEMORYPOOL, otherwise, TN_RC_WPARAM is returned.

Note that `start_addr` and `block_size` should be a multiple of `sizeof(`TN_UWord`)`.

For the definition of buffer, convenience macro TN_FMEM_BUF_DEF() was invented.

Typical definition looks as follows:

```
//-- number of blocks in the pool
#define MY_MEMORY_BUF_SIZE    8

//-- type for memory block
struct MyMemoryItem {
   // ... arbitrary fields ...
};

//-- define buffer for memory pool
TN_FMEM_BUF_DEF(my_fmem_buf, struct MyMemoryItem, MY_MEMORY_BUF_SIZE);

//-- define memory pool structure
struct TN_FMem my_fmem;
```

And then, construct your `my_fmem` as follows:

```
enum TN_RCode rc;
rc = tn_fmem_create( &my_fmem,
                     my_fmem_buf,
                     TN_MAKE_ALIG_SIZE(sizeof(struct MyMemoryItem)),
                     MY_MEMORY_BUF_SIZE );
if (rc != TN_RC_OK){
   //-- handle error
}
```

If given `start_addr` and/or `block_size` aren't aligned properly, TN_RC_WPARAM is returned.

*(refer to Legend for details)*

**Parameters**

| | |
|---:|:---|
| *fmem* | pointer to already allocated `struct` TN_FMem. |
| *start_addr* | pointer to start of the array; should be aligned properly, see example above |
| *block_size* | size of memory block; should be a multiple of `sizeof(`TN_UWord`)`, see example above |
| *blocks_cnt* | capacity (total number of blocks in the memory pool) |

**Returns**

- TN_RC_OK if memory pool was successfully created;

- If TN_CHECK_PARAM is non-zero, additional return code is available: TN_RC_WPARAM.

**See also**

TN_MAKE_ALIG_SIZE

**14.7.3.2 enum TN_RCode tn_fmem_delete ( struct TN_FMem ∗ fmem )**

Destruct fixed memory blocks pool.

All tasks that wait for free memory block become runnable with TN_RC_DELETED code returned.

 (refer to Legend for details)

**Parameters**

| | |
|---|---|
| *fmem* | pointer to memory pool to be deleted |

**Returns**

- TN_RC_OK if memory pool is successfully deleted;

- TN_RC_WCONTEXT if called from wrong context;

- If TN_CHECK_PARAM is non-zero, additional return codes are available: TN_RC_WPARAM and TN_↩
  RC_INVALID_OBJ.

**14.7.3.3 enum TN_RCode tn_fmem_get ( struct TN_FMem ∗ fmem, void ∗∗ p_data, TN_Timeout timeout )**

Get memory block from the pool.

Start address of the memory block is returned through the p_data argument. The content of memory block is undefined. If there is no free block in the pool, behavior depends on timeout value: refer to TN_Timeout.

 (refer to Legend for details)

**Parameters**

| | |
|---|---|
| *fmem* | Pointer to memory pool |
| *p_data* | Address of the (void ∗) to which received block address will be saved |
| *timeout* | Refer to TN_Timeout |

**Returns**

- TN_RC_OK if block was successfully returned through p_data;

- TN_RC_WCONTEXT if called from wrong context;

- Other possible return codes depend on timeout value, refer to TN_Timeout

- If TN_CHECK_PARAM is non-zero, additional return codes are available: TN_RC_WPARAM and TN_↩
  RC_INVALID_OBJ.

**14.7.3.4 enum TN_RCode tn_fmem_get_polling ( struct TN_FMem ∗ fmem, void ∗∗ p_data )**

The same as tn_fmem_get() with zero timeout.

 (refer to Legend for details)

**14.7.3.5 enum TN_RCode tn_fmem_iget_polling ( struct TN_FMem ∗ fmem, void ∗∗ p_data )**

The same as tn_fmem_get() with zero timeout, but for using in the ISR.

 (refer to Legend for details)

**14.7.3.6   enum TN_RCode tn_fmem_release ( struct TN_FMem ∗ *fmem,* void ∗ *p_data* )**

Release memory block back to the pool.

The kernel does not check the validity of the membership of given block in the memory pool. If all the memory blocks in the pool are free already, `TN_RC_OVERFLOW` is returned.

 *(refer to Legend for details)*

**Parameters**

| | |
|---:|---|
| *fmem* | Pointer to memory pool. |
| *p_data* | Address of the memory block to release. |

**Returns**

- `TN_RC_OK` on success

- `TN_RC_WCONTEXT` if called from wrong context;

- If `TN_CHECK_PARAM` is non-zero, additional return codes are available: `TN_RC_WPARAM` and `TN_↩
  RC_INVALID_OBJ`.

**14.7.3.7   enum TN_RCode tn_fmem_irelease ( struct TN_FMem ∗ *fmem,* void ∗ *p_data* )**

The same as `tn_fmem_get()`, but for using in the ISR.

 *(refer to Legend for details)*

# 14.8   core/tn_mutex.h File Reference

## 14.8.1   Detailed Description

A mutex is an object used to protect shared resource.

There is a lot of confusion about differences between semaphores and mutexes, so, it's quite recommended to read small article by Michael Barr: `Mutexes and Semaphores Demystified`.

Very short:

While mutex is seemingly similar to a semaphore with maximum count of `1` (the so-called binary semaphore), their usage is very different: the purpose of mutex is to protect shared resource. A locked mutex is "owned" by the task that locked it, and only the same task may unlock it. This ownership allows to implement algorithms to prevent priority inversion. So, mutex is a *locking mechanism*.

Semaphore, on the other hand, is *signaling mechanism*. It's quite legal and encouraged for semaphore to be acquired in the task A, and then signaled from task B or even from ISR. It may be used in situations like "producer and consumer", etc.

In addition to the article mentioned above, you may want to look at the `related question on stackoverflow.com`.

Mutex features in TNeoKernel:

- Recursive locking is supported (if option `TN_MUTEX_REC` is non-zero);

- Deadlock detection (if option `TN_MUTEX_DEADLOCK_DETECT` is non-zero);

- Two protocols available to avoid unbounded priority inversion: priority inheritance and priority ceiling.

A discussion about strengths and weaknesses of each protocol as well as priority inversions problem is beyond the scope of this document.

The priority inheritance protocol solves the priority inversions problem but doesn't prevents deadlocks, although the kernel can notify you if a deadlock has occured (see TN_MUTEX_DEADLOCK_DETECT).

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

**See also**

> TN_USE_MUTEXES

Definition in file tn_mutex.h.

## Data Structures

- struct TN_Mutex

    *Mutex.*

## Enumerations

- enum TN_MutexProtocol { TN_MUTEX_PROT_CEILING = 1, TN_MUTEX_PROT_INHERIT = 2 }

    *Mutex protocol for avoid priority inversion.*

## Functions

- enum TN_RCode tn_mutex_create (struct TN_Mutex ∗mutex, enum TN_MutexProtocol protocol, int ceil_↩
    priority)

    *Construct the mutex.*
- enum TN_RCode tn_mutex_delete (struct TN_Mutex ∗mutex)

    *Destruct mutex.*
- enum TN_RCode tn_mutex_lock (struct TN_Mutex ∗mutex, TN_Timeout timeout)

    *Lock mutex.*
- enum TN_RCode tn_mutex_lock_polling (struct TN_Mutex ∗mutex)

    *The same as* `tn_mutex_lock()` *with zero timeout.*
- enum TN_RCode tn_mutex_unlock (struct TN_Mutex ∗mutex)

    *Unlock mutex.*

### 14.8.2 Enumeration Type Documentation

#### 14.8.2.1 enum TN_MutexProtocol

Mutex protocol for avoid priority inversion.

**Enumerator**

> **TN_MUTEX_PROT_CEILING**  Mutex uses priority ceiling protocol.
>
> **TN_MUTEX_PROT_INHERIT**  Mutex uses priority inheritance protocol.

Definition at line 109 of file tn_mutex.h.

### 14.8.3 Function Documentation

#### 14.8.3.1 enum **TN_RCode tn_mutex_create** ( struct **TN_Mutex** ∗ *mutex,* enum **TN_MutexProtocol** *protocol,* int *ceil_priority* )

Construct the mutex.

The field `id_mutex` should not contain `TN_ID_MUTEX`, otherwise, `TN_RC_WPARAM` is returned.

🅣 🅘 *(refer to Legend for details)*

**Parameters**

| | |
|---:|---|
| *mutex* | Pointer to already allocated `struct` `TN_Mutex` |
| *protocol* | Mutex protocol: priority ceiling or priority inheritance. See `enum` `TN_MutexProtocol`. |
| *ceil_priority* | Used if only `protocol` is `TN_MUTEX_PROT_CEILING`: maximum priority of the task that may lock the mutex. |

**Returns**

- `TN_RC_OK` if mutex was successfully created;

- If `TN_CHECK_PARAM` is non-zero, additional return code is available: `TN_RC_WPARAM`.

#### 14.8.3.2 enum **TN_RCode tn_mutex_delete** ( struct **TN_Mutex** ∗ *mutex* )

Destruct mutex.

All tasks that wait for lock the mutex become runnable with `TN_RC_DELETED` code returned.

🅣 🅤 *(refer to Legend for details)*

**Parameters**

| | |
|---:|---|
| *mutex* | mutex to destruct |

**Returns**

- `TN_RC_OK` if mutex was successfully destroyed;

- `TN_RC_WCONTEXT` if called from wrong context;

- If `TN_CHECK_PARAM` is non-zero, additional return codes are available: `TN_RC_WPARAM` and `TN_↩ RC_INVALID_OBJ`.

#### 14.8.3.3 enum **TN_RCode tn_mutex_lock** ( struct **TN_Mutex** ∗ *mutex,* **TN_Timeout** *timeout* )

Lock mutex.

- If the mutex is not locked, function immediately locks the mutex and returns `TN_RC_OK`.

- If the mutex is already locked by the same task, lock count is merely incremented and `TN_RC_OK` is returned immediately.

- If the mutex is locked by different task, behavior depends on `timeout` value: refer to `TN_Timeout`.

🅣 🅤 🕐 *(refer to Legend for details)*

**Parameters**

| | |
|---:|---|
| *mutex* | mutex to lock |
| *timeout* | refer to TN_Timeout |

**Returns**

- TN_RC_OK if mutex is successfully locked or if lock count was merely incremented (this is possible if recursive locking is enabled, see TN_MUTEX_REC)

- TN_RC_WCONTEXT if called from wrong context;

- TN_RC_ILLEGAL_USE

  - if mutex protocol is TN_MUTEX_PROT_CEILING and calling task's priority is higher than ceil← _priority given to tn_mutex_create()
  - if recursive locking is disabled (see TN_MUTEX_REC) and the mutex is already locked by calling task

- Other possible return codes depend on timeout value, refer to TN_Timeout

- If TN_CHECK_PARAM is non-zero, additional return codes are available: TN_RC_WPARAM and TN_← RC_INVALID_OBJ.

**See also**

    TN_MutexProtocol

**14.8.3.4 enum TN_RCode tn_mutex_lock_polling ( struct TN_Mutex ∗ *mutex* )**

The same as tn_mutex_lock() with zero timeout.

 *(refer to Legend for details)*

**14.8.3.5 enum TN_RCode tn_mutex_unlock ( struct TN_Mutex ∗ *mutex* )**

Unlock mutex.

- If mutex is not locked or locked by different task, TN_RC_ILLEGAL_USE is returned.

- If mutex is already locked by calling task, lock count is decremented. Now, if lock count is zero, mutex gets unlocked (and if there are task(s) waiting for mutex, the first one from the wait queue locks the mutex). Otherwise, mutex remains locked with lock count decremented and function returns TN_RC_OK.

 *(refer to Legend for details)*

**Returns**

- TN_RC_OK if mutex is unlocked of if lock count was merely decremented (this is possible if recursive locking is enabled, see TN_MUTEX_REC)

- TN_RC_WCONTEXT if called from wrong context;

- TN_RC_ILLEGAL_USE if mutex is either not locked or locked by different task

- If TN_CHECK_PARAM is non-zero, additional return codes are available: TN_RC_WPARAM and TN_← RC_INVALID_OBJ.

## 14.9 core/tn_oldsymbols.h File Reference

### 14.9.1 Detailed Description

Compatibility layer for old projects that use old TNKernel names; usage of them in new projects is discouraged.

If you're porting your existing application written for TNKernel, it might be useful though.

Included automatially if the option TN_OLD_TNKERNEL_NAMES is set.

**See also**

> TN_OLD_TNKERNEL_NAMES

Definition in file tn_oldsymbols.h.

**Macros**

- #define _CDLL_QUEUE TN_ListItem

  *old TNKernel struct name of* `TN_ListItem`
- #define _TN_MUTEX TN_Mutex

  *old TNKernel struct name of* `TN_Mutex`
- #define _TN_DQUE TN_DQueue

  *old TNKernel struct name of* `TN_DQueue`
- #define _TN_TCB TN_Task

  *old TNKernel struct name of* `TN_Task`
- #define _TN_FMP TN_FMem

  *old TNKernel struct name of* `TN_FMem`
- #define _TN_SEM TN_Sem

  *old TNKernel struct name of* `TN_Sem`
- #define MAKE_ALIG TN_MAKE_ALIG

  *old TNKernel name of* `TN_MAKE_ALIG` *macro*
- #define TSK_STATE_RUNNABLE TN_TASK_STATE_RUNNABLE

  *old TNKernel name of* `TN_TASK_STATE_RUNNABLE`
- #define TSK_STATE_WAIT TN_TASK_STATE_WAIT

  *old TNKernel name of* `TN_TASK_STATE_WAIT`
- #define TSK_STATE_SUSPEND TN_TASK_STATE_SUSPEND

  *old TNKernel name of* `TN_TASK_STATE_SUSPEND`
- #define TSK_STATE_WAITSUSP TN_TASK_STATE_WAITSUSP

  *old TNKernel name of* `TN_TASK_STATE_WAITSUSP`
- #define TSK_STATE_DORMANT TN_TASK_STATE_DORMANT

  *old TNKernel name of* `TN_TASK_STATE_DORMANT`
- #define TN_TASK_START_ON_CREATION TN_TASK_CREATE_OPT_START

  *old TNKernel name of* `TN_TASK_CREATE_OPT_START`
- #define TN_EXIT_AND_DELETE_TASK TN_TASK_EXIT_OPT_DELETE

  *old TNKernel name of* `TN_TASK_EXIT_OPT_DELETE`
- #define TN_EVENT_WCOND_AND TN_EVENTGRP_WMODE_AND

  *old TNKernel name of* `TN_EVENTGRP_WMODE_AND`
- #define TN_EVENT_WCOND_OR TN_EVENTGRP_WMODE_OR

  *old TNKernel name of* `TN_EVENTGRP_WMODE_OR`
- #define TSK_WAIT_REASON_NONE TN_WAIT_REASON_NONE

  *old TNKernel name of* `TN_WAIT_REASON_NONE`
- #define TSK_WAIT_REASON_SLEEP TN_WAIT_REASON_SLEEP

*old TNKernel name of* *TN_WAIT_REASON_SLEEP*

- #define TSK_WAIT_REASON_SEM TN_WAIT_REASON_SEM

    *old TNKernel name of* *TN_WAIT_REASON_SEM*

- #define TSK_WAIT_REASON_EVENT TN_WAIT_REASON_EVENT

    *old TNKernel name of* *TN_WAIT_REASON_EVENT*

- #define TSK_WAIT_REASON_DQUE_WSEND TN_WAIT_REASON_DQUE_WSEND

    *old TNKernel name of* *TN_WAIT_REASON_DQUE_WSEND*

- #define TSK_WAIT_REASON_DQUE_WRECEIVE TN_WAIT_REASON_DQUE_WRECEIVE

    *old TNKernel name of* *TN_WAIT_REASON_DQUE_WRECEIVE*

- #define TSK_WAIT_REASON_MUTEX_C TN_WAIT_REASON_MUTEX_C

    *old TNKernel name of* *TN_WAIT_REASON_MUTEX_C*

- #define TSK_WAIT_REASON_MUTEX_I TN_WAIT_REASON_MUTEX_I

    *old TNKernel name of* *TN_WAIT_REASON_MUTEX_I*

- #define TSK_WAIT_REASON_WFIXMEM TN_WAIT_REASON_WFIXMEM

    *old TNKernel name of* *TN_WAIT_REASON_WFIXMEM*

- #define TERR_NO_ERR TN_RC_OK

    *old TNKernel name of* *TN_RC_OK*

- #define TERR_OVERFLOW TN_RC_OVERFLOW

    *old TNKernel name of* *TN_RC_OVERFLOW*

- #define TERR_WCONTEXT TN_RC_WCONTEXT

    *old TNKernel name of* *TN_RC_WCONTEXT*

- #define TERR_WSTATE TN_RC_WSTATE

    *old TNKernel name of* *TN_RC_WSTATE*

- #define TERR_TIMEOUT TN_RC_TIMEOUT

    *old TNKernel name of* *TN_RC_TIMEOUT*

- #define TERR_WRONG_PARAM TN_RC_WPARAM

    *old TNKernel name of* *TN_RC_WPARAM*

- #define TERR_ILUSE TN_RC_ILLEGAL_USE

    *old TNKernel name of* *TN_RC_ILLEGAL_USE*

- #define TERR_NOEXS TN_RC_INVALID_OBJ

    *old TNKernel name of* *TN_RC_INVALID_OBJ*

- #define TERR_DLT TN_RC_DELETED

    *old TNKernel name of* *TN_RC_DELETED*

- #define TERR_FORCED TN_RC_FORCED

    *old TNKernel name of* *TN_RC_FORCED*

- #define TERR_INTERNAL TN_RC_INTERNAL

    *old TNKernel name of* *TN_RC_INTERNAL*

- #define TN_MUTEX_ATTR_CEILING TN_MUTEX_PROT_CEILING

    *old TNKernel name of* *TN_MUTEX_PROT_CEILING*

- #define TN_MUTEX_ATTR_INHERIT TN_MUTEX_PROT_INHERIT

    *old TNKernel name of* *TN_MUTEX_PROT_INHERIT*

- #define tn_sem_polling tn_sem_acquire_polling

    *old TNKernel name of* *tn_sem_acquire_polling*

- #define tn_sem_ipolling tn_sem_iacquire_polling

    *old TNKernel name of* *tn_sem_iacquire_polling*

- #define tn_fmem_get_ipolling tn_fmem_iget_polling

    *old TNKernel name of* *tn_fmem_iget_polling*

- #define tn_queue_ireceive tn_queue_ireceive_polling

    *old TNKernel name of* *tn_queue_ireceive_polling*

- #define tn_start_system tn_sys_start

    *old TNKernel name of* *tn_sys_start*

- #define tn_sys_tslice_ticks tn_sys_tslice_set

    *old TNKernel name of* `tn_sys_tslice_set`
- #define align_attr_start TN_ARCH_STK_ATTR_BEFORE

    *old TNKernel name of* `TN_ARCH_STK_ATTR_BEFORE`
- #define align_attr_end TN_ARCH_STK_ATTR_AFTER

    *old TNKernel name of* `TN_ARCH_STK_ATTR_AFTER`
- #define tn_cpu_int_disable tn_arch_int_dis

    *old TNKernel name of* `tn_arch_int_dis`
- #define tn_cpu_int_enable tn_arch_int_en

    *old TNKernel name of* `tn_arch_int_en`
- #define tn_cpu_save_sr tn_arch_sr_save_int_dis

    *old TNKernel name of* `tn_arch_sr_save_int_dis`
- #define tn_cpu_restore_sr tn_arch_sr_restore

    *old TNKernel name of* `tn_arch_sr_restore`
- #define tn_disable_interrupt TN_INT_DIS_SAVE

    *old TNKernel name of* `TN_INT_DIS_SAVE`
- #define tn_enable_interrupt TN_INT_RESTORE

    *old TNKernel name of* `TN_INT_RESTORE`
- #define tn_idisable_interrupt TN_INT_IDIS_SAVE

    *old TNKernel name of* `TN_INT_IDIS_SAVE`
- #define tn_ienable_interrupt TN_INT_IRESTORE

    *old TNKernel name of* `TN_INT_IRESTORE`
- #define tn_chk_irq_disabled TN_IS_INT_DISABLED

    *old TNKernel name of* `TN_IS_INT_DISABLED`
- #define TN_NUM_PRIORITY TN_PRIORITIES_CNT

    *old TNKernel name of* `TN_PRIORITIES_CNT`
- #define _TN_BITS_IN_INT TN_INT_WIDTH

    *old TNKernel name of* `TN_INT_WIDTH`
- #define TN_ALIG sizeof(TN_UWord)

    *old TNKernel name for* `sizeof(TN_UWord)`


**Typedefs**

- typedef struct TN_ListItem CDLL_QUEUE

    *old TNKernel name of* `TN_ListItem`
- typedef struct TN_Mutex TN_MUTEX

    *old TNKernel name of* `TN_Mutex`
- typedef struct TN_DQueue TN_DQUE

    *old TNKernel name of* `TN_DQueue`
- typedef struct TN_Task TN_TCB

    *old TNKernel name of* `TN_Task`
- typedef struct TN_FMem TN_FMP

    *old TNKernel name of* `TN_FMem`
- typedef struct TN_Sem TN_SEM

    *old TNKernel name of* `TN_Sem`

### 14.9.2 Macro Definition Documentation

#### 14.9.2.1 #define MAKE_ALIG TN_MAKE_ALIG

old TNKernel name of `TN_MAKE_ALIG` macro

**Attention**

> it is recommended to use `TN_MAKE_ALIG_SIZE` macro instead of this one, in order to avoid confusion caused by various TNKernel ports: refer to the section Macro MAKE_ALIG() for details.

Definition at line 142 of file tn_oldsymbols.h.

## 14.10 core/tn_sem.h File Reference

### 14.10.1 Detailed Description

A semaphore: an object to provide signaling mechanism.

There is a lot of confusion about differences between semaphores and mutexes, so, it's quite recommended to read small article by Michael Barr: `Mutexes and Semaphores Demystified`.

Very short:

While mutex is seemingly similar to a semaphore with maximum count of `1` (the so-called binary semaphore), their usage is very different: the purpose of mutex is to protect shared resource. A locked mutex is "owned" by the task that locked it, and only the same task may unlock it. This ownership allows to implement algorithms to prevent priority inversion. So, mutex is a *locking mechanism*.

Semaphore, on the other hand, is *signaling mechanism*. It's quite legal and encouraged for semaphore to be acquired in the task A, and then signaled from task B or even from ISR. It may be used in situations like "producer and consumer", etc.

In addition to the article mentioned above, you may want to look at the `related question on stackoverflow.com`.

Definition in file tn_sem.h.

**Data Structures**

- struct TN_Sem

    *Semaphore.*

**Functions**

- enum TN_RCode tn_sem_create (struct TN_Sem ∗sem, int start_count, int max_count)

    *Construct the semaphore.*
- enum TN_RCode tn_sem_delete (struct TN_Sem ∗sem)

    *Destruct the semaphore.*
- enum TN_RCode tn_sem_signal (struct TN_Sem ∗sem)

    *Signal the semaphore.*
- enum TN_RCode tn_sem_isignal (struct TN_Sem ∗sem)

    *The same as `tn_sem_signal()` but for using in the ISR.*
- enum TN_RCode tn_sem_acquire (struct TN_Sem ∗sem, TN_Timeout timeout)

    *Acquire the semaphore.*
- enum TN_RCode tn_sem_acquire_polling (struct TN_Sem ∗sem)

*The same as* *tn_sem_acquire()* *with zero timeout.*

- enum TN_RCode tn_sem_iacquire_polling (struct TN_Sem ∗sem)

  *The same as* *tn_sem_acquire()* *with zero timeout, but for using in the ISR.*

## 14.10.2 Function Documentation

### 14.10.2.1 enum **TN_RCode tn_sem_create ( struct TN_Sem** ∗ *sem,* **int** *start_count,* **int** *max_count* **)**

Construct the semaphore.

id_sem field should not contain TN_ID_SEMAPHORE, otherwise, TN_RC_WPARAM is returned.

 *(refer to Legend for details)*

**Parameters**

| | |
|---|---|
| *sem* | Pointer to already allocated struct TN_Sem |
| *start_count* | Initial counter value, typically it is equal to max_count |
| *max_count* | Maximum counter value. |

**Returns**

- TN_RC_OK if semaphore was successfully created;

- If TN_CHECK_PARAM is non-zero, additional return code is available: TN_RC_WPARAM.

### 14.10.2.2 enum **TN_RCode tn_sem_delete ( struct TN_Sem** ∗ *sem* **)**

Destruct the semaphore.

All tasks that wait for acquire the semaphore become runnable with TN_RC_DELETED code returned.

 *(refer to Legend for details)*

**Parameters**

| | |
|---|---|
| *sem* | semaphore to destruct |

**Returns**

- TN_RC_OK if semaphore was successfully deleted;

- TN_RC_WCONTEXT if called from wrong context;

- If TN_CHECK_PARAM is non-zero, additional return codes are available: TN_RC_WPARAM and TN_↩
  RC_INVALID_OBJ.

### 14.10.2.3 enum **TN_RCode tn_sem_signal ( struct TN_Sem** ∗ *sem* **)**

Signal the semaphore.

If current semaphore counter (count) is less than max_count, counter is incremented by one; otherwise, TN↩
_RC_OVERFLOW is returned.

If wait queue is not empty, the first task from the queue acquires the semaphore.

 *(refer to Legend for details)*

**Parameters**

| | |
|---:|---|
| *sem* | semaphore to signal |

**Returns**

- [TN_RC_OK](#) if successful

- [TN_RC_WCONTEXT](#) if called from wrong context;

- [TN_RC_OVERFLOW](#) if `count` is already at maximum value (`max_count`)

- If [TN_CHECK_PARAM](#) is non-zero, additional return codes are available: [TN_RC_WPARAM](#) and [TN_←RC_INVALID_OBJ](#).

**14.10.2.4  enum TN_RCode tn_sem_isignal ( struct TN_Sem ∗ *sem* )**

The same as [tn_sem_signal()](#) but for using in the ISR.

*(refer to Legend for details)*

**14.10.2.5  enum TN_RCode tn_sem_acquire ( struct TN_Sem ∗ *sem,* TN_Timeout *timeout* )**

Acquire the semaphore.

If the current semaphore counter (`count`) is non-zero, it is decremented and [TN_RC_OK](#) is returned. Otherwise, behavior depends on `timeout` value: refer to [TN_Timeout](#).

*(refer to Legend for details)*

**Parameters**

| | |
|---:|---|
| *sem* | semaphore to acquire |
| *timeout* | refer to [TN_Timeout](#) |

**Returns**

- [TN_RC_OK](#) if semaphore was successfully acquired

- Other possible return codes depend on `timeout` value, refer to [TN_Timeout](#)

- If [TN_CHECK_PARAM](#) is non-zero, additional return codes are available: [TN_RC_WPARAM](#) and [TN_←RC_INVALID_OBJ](#).

**14.10.2.6  enum TN_RCode tn_sem_acquire_polling ( struct TN_Sem ∗ *sem* )**

The same as [tn_sem_acquire()](#) with zero timeout.

*(refer to Legend for details)*

**14.10.2.7  enum TN_RCode tn_sem_iacquire_polling ( struct TN_Sem ∗ *sem* )**

The same as [tn_sem_acquire()](#) with zero timeout, but for using in the ISR.

*(refer to Legend for details)*

## 14.11 core/tn_sys.h File Reference

### 14.11.1 Detailed Description

Kernel system routines: system start, tick processing, time slice managing.

Definition in file tn_sys.h.

### Macros

- #define TN_NO_TIME_SLICE 0

  *Value to pass to `tn_sys_tslice_set()` to turn round-robin off.*
- #define TN_MAX_TIME_SLICE 0xFFFE

  *Max value of time slice.*

### Typedefs

- typedef void( TN_CBUserTaskCreate )(void)

  *User-provided callback function that is called directly from `tn_sys_start()` as a part of system startup routine; it should merely create at least one (and typically just one) user's task, which should perform all the rest application initialization.*
- typedef void( TN_CBIdle )(void)

  *User-provided callback function that is called repeatedly from the idle task loop.*
- typedef void( TN_CBDeadlock )(BOOL active, struct TN_Mutex ∗mutex, struct TN_Task ∗task)

  *User-provided callback function that is called whenever deadlock becomes active or inactive.*

### Enumerations

- enum TN_StateFlag { TN_STATE_FLAG__SYS_RUNNING = (1 << 0), TN_STATE_FLAG__DEADLOCK = (1 << 1) }

  *System state flags.*
- enum TN_Context { TN_CONTEXT_NONE, TN_CONTEXT_TASK, TN_CONTEXT_ISR }

  *System context.*

### Functions

- void tn_sys_start (TN_UWord ∗idle_task_stack, unsigned int idle_task_stack_size, TN_UWord ∗int_stack, unsigned int int_stack_size, TN_CBUserTaskCreate ∗cb_user_task_create, TN_CBIdle ∗cb_idle)

  *Initial TNeoKernel system start function, never returns.*
- enum TN_RCode tn_tick_int_processing (void)

  *Process system tick; should be called periodically, typically from some kind of timer ISR.*
- enum TN_RCode tn_sys_tslice_set (int priority, int ticks)

  *Set time slice ticks value for specified priority (see Round-robin scheduling).*
- unsigned int tn_sys_time_get (void)

  *Get current system ticks count.*
- void tn_sys_time_set (unsigned int value)

  *Set system ticks count.*
- void tn_callback_deadlock_set (TN_CBDeadlock ∗cb)

  *Set callback function that should be called whenever deadlock occurs or becomes inactive (say, if one of tasks involved in the deadlock was released from wait because of timeout)*
- enum TN_StateFlag tn_sys_state_flags_get (void)

*Returns current system state flags.*

- enum TN_Context tn_sys_context_get (void)

  *Returns system context: task or ISR.*

- static BOOL tn_is_task_context (void)

  *Returns whether current system context is* `TN_CONTEXT_TASK`

- static BOOL tn_is_isr_context (void)

  *Returns whether current system context is* `TN_CONTEXT_ISR`

- struct TN_Task ∗ tn_cur_task_get (void)

  *Returns pointer to the currently running task.*

- TN_TaskBody ∗ tn_cur_task_body_get (void)

  *Returns pointer to the body function of the currently running task.*


### 14.11.2 Typedef Documentation

#### 14.11.2.1 typedef void( TN_CBUserTaskCreate)(void)

User-provided callback function that is called directly from `tn_sys_start()` as a part of system startup routine; it should merely create at least one (and typically just one) user's task, which should perform all the rest application initialization.

When `TN_CBUserTaskCreate()` returned, the kernel performs first context switch to the task with highest priority. If there are several tasks with highest priority, context is switched to the first created one.

Refer to the section Starting the kernel for details about system startup process on the whole.

**Note:** Although you're able to create more than one task here, it's usually not so good idea, because many things typically should be done at startup before tasks can go on with their job: we need to initialize various on-board peripherals (displays, flash memory chips, or whatever) as well as initialize software modules used by application. So, if many tasks are created here, you have to provide some synchronization object so that tasks will wait until all the initialization is done.

It's usually easier to maintain if we create just one task here, which firstly performs all the necessary initialization, **then** creates the rest of your tasks, and eventually gets to its primary job (the job for which task was created at all). For the usage example, refer to the page Starting the kernel.

**Attention**

- The only system service is allowed to call in this function is `tn_task_create()`.


**See also**

   `tn_sys_start()`

Definition at line 141 of file tn_sys.h.


#### 14.11.2.2 typedef void( TN_CBIdle)(void)

User-provided callback function that is called repeatedly from the idle task loop.

Make sure that idle task has enough stack space to call this function.

**Attention**

- It is illegal to sleep here, because idle task (from which this function is called) should always be runnable, by design. If `TN_DEBUG` option is set, then sleeping in idle task is checked, so if you try to sleep here, `_TN_FATAL_ERROR()` macro will be called.

**See also**

    tn_sys_start()

Definition at line 156 of file tn_sys.h.

**14.11.2.3  typedef void( TN_CBDeadlock)(BOOL active, struct TN_Mutex ∗mutex, struct TN_Task ∗task)**

User-provided callback function that is called whenever deadlock becomes active or inactive.

Note: this feature works if only TN_MUTEX_DEADLOCK_DETECT is non-zero.

**Parameters**

| | |
|---:|---|
| *active* | if TRUE, deadlock becomes active, otherwise it becomes inactive (say, if task stopped waiting for mutex because of timeout) |
| *mutex* | mutex that is involved in deadlock. You may find out other mutexes involved by means of mutex->deadlock_list. |
| *task* | task that is involved in deadlock. You may find out other tasks involved by means of task->deadlock_list. |

Definition at line 171 of file tn_sys.h.

## 14.11.3  Enumeration Type Documentation

**14.11.3.1  enum TN_StateFlag**

System state flags.

**Enumerator**

    ***TN_STATE_FLAG__SYS_RUNNING***  system is running

    ***TN_STATE_FLAG__DEADLOCK***  deadlock is active Note: this feature works if only TN_MUTEX_DEADLO↩
CK_DETECT is non-zero.

        **See also**

            TN_MUTEX_DEADLOCK_DETECT

Definition at line 79 of file tn_sys.h.

**14.11.3.2  enum TN_Context**

System context.

**See also**

    tn_sys_context_get()

**Enumerator**

    ***TN_CONTEXT_NONE***  None: this code is possible if only system is not running (flag (TN_STATE_FLAG↩
_SYS_RUNNING is not set in the tn_sys_state))

    ***TN_CONTEXT_TASK***  Task context.

    ***TN_CONTEXT_ISR***  ISR context.

Definition at line 95 of file tn_sys.h.

### 14.11.4 Function Documentation

#### 14.11.4.1 void tn_sys_start ( TN_UWord ∗ *idle_task_stack,* unsigned int *idle_task_stack_size,* TN_UWord ∗ *int_stack,* unsigned int *int_stack_size,* TN_CBUserTaskCreate ∗ *cb_user_task_create,* TN_CBIdle ∗ *cb_idle* )

Initial TNeoKernel system start function, never returns.

Typically called from main().

Refer to the Starting the kernel section for the usage example and additional comments.

*(refer to Legend for details)*

**Parameters**

| | |
|---|---|
| *idle_task_stack* | Pointer to array for idle task stack. User must either use the macro `TN_TASK_STACK_D↵EF()` for the definition of stack array, or allocate it manually as an array of `TN_UWord` with `TN_ARCH_STK_ATTR_BEFORE` and `TN_ARCH_STK_ATTR_AFTER` macros. |
| *idle_task_↵stack_size* | Size of idle task stack, in words (`TN_UWord`) |
| *int_stack* | Pointer to array for interrupt stack. User must either use the macro `TN_TASK_STACK_D↵EF()` for the definition of stack array, or allocate it manually as an array of `TN_UWord` with `TN_ARCH_STK_ATTR_BEFORE` and `TN_ARCH_STK_ATTR_AFTER` macros. |
| *int_stack_size* | Size of interrupt stack, in words (`TN_UWord`) |
| *cb_user_task_↵create* | Callback function that should create initial user's task (see TN_CBUserTaskCreate) |
| *cb_idle* | Callback function repeatedly called from idle task. |

#### 14.11.4.2 enum **TN_RCode** tn_tick_int_processing ( void )

Process system tick; should be called periodically, typically from some kind of timer ISR.

The period of this timer is determined by user (typically 1 ms, but user is free to set different value)

For further information, refer to Quick guide.

*(refer to Legend for details)*

**Returns**

- `TN_RC_OK` on success;
- `TN_RC_WCONTEXT` if called from wrong context.

#### 14.11.4.3 enum **TN_RCode** tn_sys_tslice_set ( int *priority,* int *ticks* )

Set time slice ticks value for specified priority (see Round-robin scheduling).

*(refer to Legend for details)*

**Parameters**

| | |
|---|---|
| *priority* | Priority of tasks for which time slice value should be set |
| *ticks* | Time slice value, in ticks. Set to `TN_NO_TIME_SLICE` for no round-robin scheduling for given priority (it's default value). Value can't be higher than `TN_MAX_TIME_SLICE`. |

**Returns**

- `TN_RC_OK` on success;
- `TN_RC_WCONTEXT` if called from wrong context;
- `TN_RC_WPARAM` if given `priority` or `ticks` are invalid.

**14.11.4.4   unsigned int tn_sys_time_get ( void )**

Get current system ticks count.

 *(refer to Legend for details)*
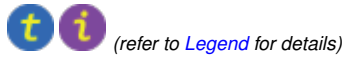
**Returns**

> Current system ticks count. Note that this value does **not** affect any of the internal TNeoKernel routines, it is just incremented each system tick (i.e. in `tn_tick_int_processing()`) and is returned to user by `tn_sys_time_get()`.

It is not used by TNeoKernel itself at all.

**14.11.4.5   void tn_sys_time_set ( unsigned int *value* )**

Set system ticks count.

Note that this value does **not** affect any of the internal TNeoKernel routines, it is just incremented each system tick (i.e. in `tn_tick_int_processing()`) and is returned to user by `tn_sys_time_get()`.

It is not used by TNeoKernel itself at all.

 *(refer to Legend for details)*
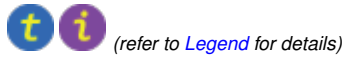
**14.11.4.6   void tn_callback_deadlock_set ( TN_CBDeadlock ∗ *cb* )**

Set callback function that should be called whenever deadlock occurs or becomes inactive (say, if one of tasks involved in the deadlock was released from wait because of timeout)

*(refer to Legend for details)*

**Note:** this function should be called before `tn_sys_start()`

**Parameters**

| | |
|---:|---|
| *cb* | Pointer to user-provided callback function. |

**See also**

> `TN_MUTEX_DEADLOCK_DETECT`
> `TN_CBDeadlock` for callback function prototype

**14.11.4.7   enum TN_StateFlag tn_sys_state_flags_get ( void )**

Returns current system state flags.

 *(refer to Legend for details)*
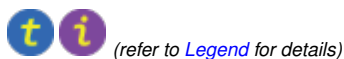
**14.11.4.8   enum TN_Context tn_sys_context_get ( void )**

Returns system context: task or ISR.

 *(refer to Legend for details)*

**See also**

> enum `TN_Context`

**14.11.4.9   static BOOL tn_is_task_context ( void )**  `[inline],[static]`

Returns whether current system context is TN_CONTEXT_TASK

*(refer to Legend for details)*

**Returns**

> `TRUE` if current system context is TN_CONTEXT_TASK, `FALSE` otherwise.

**See also**

> `tn_sys_context_get()`
> enum TN_Context

Definition at line 364 of file tn_sys.h.

**14.11.4.10   static BOOL tn_is_isr_context ( void )**  `[inline],[static]`

Returns whether current system context is TN_CONTEXT_ISR

*(refer to Legend for details)*

**Returns**

> `TRUE` if current system context is TN_CONTEXT_ISR, `FALSE` otherwise.

**See also**

> `tn_sys_context_get()`
> enum TN_Context

Definition at line 383 of file tn_sys.h.

**14.11.4.11   struct TN_Task∗ tn_cur_task_get ( void )**

Returns pointer to the currently running task.

*(refer to Legend for details)*

**14.11.4.12   TN_TaskBody∗ tn_cur_task_body_get ( void )**

Returns pointer to the body function of the currently running task.

*(refer to Legend for details)*

## 14.12   core/tn_tasks.h File Reference

### 14.12.1   Detailed Description

Various task services: create, sleep, wake up, terminate, etc.

Definition in file tn_tasks.h.

## Data Structures

- struct TN_Task

    *Task.*

## Macros

- #define TN_TASK_STACK_DEF(name, size)

    *Convenience macro for the definition of stack array.*

## Enumerations

- enum TN_TaskState {
  TN_TASK_STATE_NONE = 0, TN_TASK_STATE_RUNNABLE = (1 << 0), TN_TASK_STATE_WAIT = (1 << 1), TN_TASK_STATE_SUSPEND = (1 << 2),
  TN_TASK_STATE_WAITSUSP = (TN_TASK_STATE_WAIT | TN_TASK_STATE_SUSPEND), TN_TASK↩
  _STATE_DORMANT = (1 << 3) }

    *Task state.*
- enum TN_WaitReason {
  TN_WAIT_REASON_NONE, TN_WAIT_REASON_SLEEP, TN_WAIT_REASON_SEM, TN_WAIT_REAS↩
  ON_EVENT,
  TN_WAIT_REASON_DQUE_WSEND, TN_WAIT_REASON_DQUE_WRECEIVE, TN_WAIT_REASON_M↩
  UTEX_C, TN_WAIT_REASON_MUTEX_I,
  TN_WAIT_REASON_WFIXMEM }

    *Task wait reason.*
- enum TN_TaskCreateOpt { TN_TASK_CREATE_OPT_START = (1 << 0), TN_TASK_CREATE_OPT_IDLE = (1 << 1) }

    *Options for* `tn_task_create()`
- enum TN_TaskExitOpt { TN_TASK_EXIT_OPT_DELETE = (1 << 0) }

    *Options for* `tn_task_exit()`

## Functions

- enum TN_RCode tn_task_create (struct TN_Task ∗task, TN_TaskBody ∗task_func, int priority, TN_UWord
  ∗task_stack_low_addr, int task_stack_size, void ∗param, enum TN_TaskCreateOpt opts)

    *Construct task and probably start it (depends on options, see below).*
- enum TN_RCode tn_task_suspend (struct TN_Task ∗task)

    *If the task is* RUNNABLE, *it is moved to the* SUSPEND *state.*
- enum TN_RCode tn_task_resume (struct TN_Task ∗task)

    *Release task from* SUSPEND *state.*
- enum TN_RCode tn_task_sleep (TN_Timeout timeout)

    *Put current task to sleep for at most timeout ticks.*
- enum TN_RCode tn_task_wakeup (struct TN_Task ∗task)

    *Wake up task from sleep.*
- enum TN_RCode tn_task_iwakeup (struct TN_Task ∗task)

    *The same as* `tn_task_wakeup()` *but for using in the ISR.*
- enum TN_RCode tn_task_activate (struct TN_Task ∗task)

    *Activate task that is in* DORMANT *state, that is, it was either just created by* `tn_task_create()` *without* `TN_T↩`
    `ASK_CREATE_OPT_START` *option, or terminated.*
- enum TN_RCode tn_task_iactivate (struct TN_Task ∗task)

    *The same as* `tn_task_activate()` *but for using in the ISR.*
- enum TN_RCode tn_task_release_wait (struct TN_Task ∗task)

*Release task from `WAIT` state, independently of the reason of waiting.*

- enum TN_RCode tn_task_irelease_wait (struct TN_Task ∗task)

  *The same as `tn_task_release_wait()` but for using in the ISR.*

- void tn_task_exit (enum TN_TaskExitOpt opts)

  *This function terminates the currently running task.*

- enum TN_RCode tn_task_terminate (struct TN_Task ∗task)

  *This function is similar to `tn_task_exit()` but it terminates any task other than currently running one.*

- enum TN_RCode tn_task_delete (struct TN_Task ∗task)

  *This function deletes the task specified by the task.*

- enum TN_RCode tn_task_change_priority (struct TN_Task ∗task, int new_priority)

  *Set new priority for task.*

## 14.12.2 Macro Definition Documentation

### 14.12.2.1 #define TN_TASK_STACK_DEF( *name, size* )

**Value:**

```
TN_ARCH_STK_ATTR_BEFORE                          \
   TN_UWord name[ (size) ]                       \
   TN_ARCH_STK_ATTR_AFTER
```

Convenience macro for the definition of stack array.

See `tn_task_create()` for the usage example.

**Parameters**

| | |
|---:|---|
| *name* | C variable name of the array |
| *size* | size of the stack array in words (TN_UWord), not in bytes. |

Definition at line 294 of file tn_tasks.h.

## 14.12.3 Enumeration Type Documentation

### 14.12.3.1 enum TN_TaskState

Task state.

**Enumerator**

**TN_TASK_STATE_NONE** This state should never be publicly available. It may be stored in task_state only temporarily, while some system service is in progress.

**TN_TASK_STATE_RUNNABLE** Task is ready to run (it doesn't mean that it is running at the moment)

**TN_TASK_STATE_WAIT** Task is waiting. The reason of waiting can be obtained from `task_wait_↩ reason` field of the `struct TN_Task`.

**See also**

  enum TN_WaitReason

**TN_TASK_STATE_SUSPEND** Task is suspended (by some other task)

**TN_TASK_STATE_WAITSUSP** Task was previously waiting, and after this it was suspended.

**TN_TASK_STATE_DORMANT** Task isn't yet activated or it was terminated by `tn_task_terminate()`.

Definition at line 72 of file tn_tasks.h.

**14.12.3.2 enum TN_WaitReason**

Task wait reason.

**Enumerator**

>  ***TN_WAIT_REASON_NONE*** task isn't waiting for anything
>
>  ***TN_WAIT_REASON_SLEEP*** task has called `tn_task_sleep()`
>
>  ***TN_WAIT_REASON_SEM*** task waits to acquire a semaphore
>
>  > **See also**
>  >
>  > > tn_sem.h
>
>  ***TN_WAIT_REASON_EVENT*** task waits for some event in the event group to be set
>
>  > **See also**
>  >
>  > > tn_eventgrp.h
>
>  ***TN_WAIT_REASON_DQUE_WSEND*** task wants to put some data to the data queue, and there's no space in the queue.
>
>  > **See also**
>  >
>  > > tn_dqueue.h
>
>  ***TN_WAIT_REASON_DQUE_WRECEIVE*** task wants to receive some data to the data queue, and there's no data in the queue
>
>  > **See also**
>  >
>  > > tn_dqueue.h
>
>  ***TN_WAIT_REASON_MUTEX_C*** task wants to lock a mutex with priority ceiling
>
>  > **See also**
>  >
>  > > tn_mutex.h
>
>  ***TN_WAIT_REASON_MUTEX_I*** task wants to lock a mutex with priority inheritance
>
>  > **See also**
>  >
>  > > tn_mutex.h
>
>  ***TN_WAIT_REASON_WFIXMEM*** task wants to get memory block from memory pool, and there's no free memory blocks
>
>  > **See also**
>  >
>  > > tn_fmem.h

Definition at line 101 of file tn_tasks.h.

**14.12.3.3 enum TN_TaskCreateOpt**

Options for `tn_task_create()`

**Enumerator**

>  ***TN_TASK_CREATE_OPT_START*** whether task should be activated right after it is created. If this flag is not set, user must activate task manually by calling `tn_task_activate()`.
>
>  ***TN_TASK_CREATE_OPT_IDLE*** for internal kernel usage only: this option must be provided when creating idle task

Definition at line 144 of file tn_tasks.h.

**14.12.3.4 enum TN_TaskExitOpt**

Options for `tn_task_exit()`

**Enumerator**

>   ***TN_TASK_EXIT_OPT_DELETE*** whether task should be deleted right after it is exited. If this flag is not set, user must either delete it manually by calling `tn_task_delete()` or re-activate it by calling `tn_↩ task_activate()`.

Definition at line 159 of file tn_tasks.h.

## 14.12.4 Function Documentation

**14.12.4.1 enum TN_RCode tn_task_create ( struct TN_Task ∗ *task,* TN_TaskBody ∗ *task_func,* int *priority,* TN_UWord ∗ *task_stack_low_addr,* int *task_stack_size,* void ∗ *param,* enum TN_TaskCreateOpt *opts* )**

Construct task and probably start it (depends on options, see below).

`id_task` member should not contain `TN_ID_TASK`, otherwise, `TN_RC_WPARAM` is returned.

Usage example:

```
#define MY_TASK_STACK_SIZE   (TN_MIN_STACK_SIZE + 200)
#define MY_TASK_PRIORITY     5

struct TN_Task my_task;

//-- define stack array, we use convenience macro TN_TASK_STACK_DEF()
//   for that
TN_TASK_STACK_DEF(my_task_stack, MY_TASK_STACK_SIZE);

void my_task_body(void *param)
{
   //-- an endless loop
   for (;;){
      tn_task_sleep(1);

      //-- probably do something useful
   }
}
```

And then, somewhere from other task or from the callback `TN_CBUserTaskCreate` given to `tn_sys_↩ start()`:

```
enum TN_RCode rc = tn_task_create(
      &my_task,
      my_task_body,
      MY_TASK_PRIORITY,
      my_task_stack,
      MY_TASK_STACK_SIZE,
      NULL,                    //-- parameter isn't used
      TN_TASK_CREATE_OPT_START //-- start task on creation
      );

if (rc != TN_RC_OK){
   //-- handle error
}
```

*(refer to Legend for details)*

**Parameters**

| | |
|---|---|
| *task* | Ready-allocated `struct TN_Task` structure. `id_task` member should not contain `T↩ N_ID_TASK`, otherwise `TN_RC_WPARAM` is returned. |

| | |
|---:|---|
| *task_func* | Pointer to task body function. |
| *priority* | Priority for new task. **NOTE**: the lower value, the higher priority. Must be $> 0$ and $<$ (TN_↩ PRIORITIES_CNT − 1). |
| *task_stack_↩ low_addr* | Pointer to the stack for task. User must either use the macro TN_TASK_STACK_DEF() for the definition of stack array, or allocate it manually as an array of TN_UWord with TN_AR↩ CH_STK_ATTR_BEFORE and TN_ARCH_STK_ATTR_AFTER macros. |
| *task_stack_size* | Size of task stack array, in words (TN_UWord), not in bytes. |
| *param* | Parameter that is passed to task_func. |
| *opts* | Options for task creation, refer to enum TN_TaskCreateOpt |

**Returns**

- TN_RC_OK on success;

- TN_RC_WCONTEXT if called from wrong context;

- TN_RC_WPARAM if wrong params were given;

**See also**

  TN_ARCH_STK_ATTR_BEFORE
  TN_ARCH_STK_ATTR_AFTER

**14.12.4.2   enum TN_RCode tn_task_suspend ( struct TN_Task ∗ *task* )**

If the task is RUNNABLE, it is moved to the SUSPEND state.

If the task is in the WAIT state, it is moved to the WAIT+SUSPEND state. (waiting + suspended)

 *(refer to Legend for details)*

**Parameters**

| | |
|---:|---|
| *task* | Task to suspend |

**Returns**

- TN_RC_OK on success;

- TN_RC_WCONTEXT if called from wrong context;

- TN_RC_WSTATE if task is already suspended or dormant;

- If TN_CHECK_PARAM is non-zero, additional return codes are available: TN_RC_WPARAM and TN_↩ RC_INVALID_OBJ.

**See also**

  enum TN_TaskState

**14.12.4.3   enum TN_RCode tn_task_resume ( struct TN_Task ∗ *task* )**

Release task from SUSPEND state.

If the given task is in the SUSPEND state, it is moved to RUNNABLE state; afterwards it has the lowest precedence among runnable tasks with the same priority. If the task is in WAIT+SUSPEND state, it is moved to WAIT state.

 *(refer to Legend for details)*

**Parameters**

| | |
|---|---|
| *task* | Task to release from suspended state |

**Returns**

- TN_RC_OK on success;

- TN_RC_WCONTEXT if called from wrong context;

- TN_RC_WSTATE if task is not suspended;

- If TN_CHECK_PARAM is non-zero, additional return codes are available: TN_RC_WPARAM and TN_←
  RC_INVALID_OBJ.

**See also**

> enum TN_TaskState

### 14.12.4.4 enum **TN_RCode** tn_task_sleep ( **TN_Timeout** *timeout* )

Put current task to sleep for at most timeout 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.

 *(refer to Legend for details)*

**Parameters**

| | |
|---|---|
| *timeout* | Refer to TN_Timeout |

**Returns**

- TN_RC_TIMEOUT if task has slept specified timeout;

- TN_RC_OK if task was woken up from other task by tn_task_wakeup()

- TN_RC_FORCED if task was released from wait forcibly by tn_task_release_wait()

- TN_RC_WCONTEXT if called from wrong context

**See also**

> TN_Timeout

### 14.12.4.5 enum **TN_RCode** tn_task_wakeup ( struct **TN_Task** ∗ *task* )

Wake up task from sleep.

Task is woken up if only it sleeps because of call to tn_task_sleep(). If task sleeps for some another reason, task won't be woken up, and tn_task_wakeup() returns TN_RC_WSTATE.

After this call, tn_task_sleep() returns TN_RC_OK.

 *(refer to Legend for details)*

**Parameters**

| | | |
|---|---|---|
| *task* | sleeping task to wake up | |

**Returns**

- TN_RC_OK if successful

- TN_RC_WSTATE if task is not sleeping, or it is sleeping for some reason other than tn_task_↩ sleep() call.

- TN_RC_WCONTEXT if called from wrong context;

- If TN_CHECK_PARAM is non-zero, additional return codes are available: TN_RC_WPARAM and TN_↩ RC_INVALID_OBJ.

**14.12.4.6 enum TN_RCode tn_task_iwakeup ( struct TN_Task ∗ *task* )**

The same as tn_task_wakeup() but for using in the ISR.

*(refer to Legend for details)*

**14.12.4.7 enum TN_RCode tn_task_activate ( struct TN_Task ∗ *task* )**

Activate task that is in DORMANT state, that is, it was either just created by tn_task_create() without TN_↩ TASK_CREATE_OPT_START option, or terminated.

Task is moved from DORMANT state to the RUNNABLE state.

*(refer to Legend for details)*

**Parameters**

| | | |
|---|---|---|
| *task* | dormant task to activate | |

**Returns**

- TN_RC_OK if successful

- TN_RC_WSTATE if task is not dormant

- TN_RC_WCONTEXT if called from wrong context;

- If TN_CHECK_PARAM is non-zero, additional return codes are available: TN_RC_WPARAM and TN_↩ RC_INVALID_OBJ.

**See also**

TN_TaskState

**14.12.4.8 enum TN_RCode tn_task_iactivate ( struct TN_Task ∗ *task* )**

The same as tn_task_activate() but for using in the ISR.

*(refer to Legend for details)*

**14.12.4.9   enum TN_RCode tn_task_release_wait ( struct TN_Task ∗ *task* )**

Release task from WAIT state, independently of the reason of waiting.

If task is in WAIT state, it is moved to RUNNABLE state. If task is in WAIT+SUSPEND state, it is moved to SUSPEND state.

TN_RC_FORCED is returned to the waiting task.

 *(refer to Legend for details)*

**Attention**

> Usage of this function is discouraged, since the need for it indicates bad software design

**Parameters**

| | |
|---:|---|
| *task* | task waiting for anything |

**Returns**

> - TN_RC_OK if successful
> - TN_RC_WSTATE if task is not waiting for anything
> - TN_RC_WCONTEXT if called from wrong context;
> - If TN_CHECK_PARAM is non-zero, additional return codes are available: TN_RC_WPARAM and TN_↩
>   RC_INVALID_OBJ.

**See also**

> TN_TaskState

**14.12.4.10   enum TN_RCode tn_task_irelease_wait ( struct TN_Task ∗ *task* )**

The same as tn_task_release_wait() but for using in the ISR.

 *(refer to Legend for details)*

**14.12.4.11   void tn_task_exit ( enum TN_TaskExitOpt *opts* )**

This function terminates the currently running task.

The task is moved to the DORMANT state.

After exiting, the task may be either deleted by the tn_task_delete() function call or reactivated by the tn_↩
task_activate() / tn_task_iactivate() 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
RUNNABLE state.

If this function is invoked with TN_TASK_EXIT_OPT_DELETE option set, the task will be deleted after termination
and cannot be reactivated (needs recreation).

 *(refer to Legend for details)*

**Returns**

> Returns if only called from wrong context. Normally, it never returns (since calling task becomes terminated)

**See also**

> TN_TASK_EXIT_OPT_DELETE
> tn_task_delete()
> tn_task_activate()
> tn_task_iactivate()

### 14.12.4.12    enum **TN_RCode** tn_task_terminate (  struct **TN_Task** ∗ *task* )

This function is similar to tn_task_exit() but it terminates any task other than currently running one.

After task is terminated, the task may be either deleted by the tn_task_delete() function call or reactivated by the tn_task_activate() / tn_task_iactivate() 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 RUNNABLE state.

*(refer to Legend for details)*

**Parameters**

| | |
|---|---|
| *task* | task to terminate |

**Returns**

- TN_RC_OK if successful

- TN_RC_WSTATE if task is already dormant

- TN_RC_WCONTEXT if called from wrong context;

- If TN_CHECK_PARAM is non-zero, additional return codes are available: TN_RC_WPARAM and TN_↩
  RC_INVALID_OBJ.

### 14.12.4.13    enum **TN_RCode** tn_task_delete (  struct **TN_Task** ∗ *task* )

This function deletes the task specified by the task.

The task must be in the DORMANT state, otherwise TN_RC_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).

*(refer to Legend for details)*

**Parameters**

| | |
|---|---|
| *task* | dormant task to delete |

**Returns**

- TN_RC_OK if successful

- TN_RC_WSTATE if task is not dormant

- TN_RC_WCONTEXT if called from wrong context;

- If TN_CHECK_PARAM is non-zero, additional return codes are available: TN_RC_WPARAM and TN_↩
  RC_INVALID_OBJ.

**14.12.4.14   enum TN_RCode tn_task_change_priority ( struct TN_Task * *task,* int *new_priority* )**

Set new priority for task.

If priority is 0, then task's base_priority is set.

![t](refer to *Legend* for details)

**Attention**

this function is obsolete and will probably be removed

## 14.13   tn.h File Reference

### 14.13.1   Detailed Description

The main kernel header file that should be included by user application; it merely includes subsystem-specific kernel headers.

Definition in file tn.h.

## 14.14   tn_cfg_default.h File Reference

### 14.14.1   Detailed Description

TNeoKernel default configuration file, to be copied as `tn_cfg.h`.

This project is intended to be built as a library, separately from main project (although nothing prevents you from bundling things together, if you want to).

There are various options available which affects API and behavior of the kernel. But these options are specific for particular project, and aren't related to the kernel itself, so we need to keep them separately.

To this end, file `tn.h` (the main kernel header file) includes `tn_cfg.h`, which isn't included in the repository (even more, it is added to `.hgignore` list actually). Instead, default configuration file `tn_cfg_default.h` is provided, and when you just cloned the repository, you might want to copy it as `tn_cfg.h`. Or even better, if your filesystem supports symbolic links, copy it somewhere to your main project's directory (so that you can add it to your VCS there), and create symlink to it named `tn_cfg.h` in the TNeoKernel source directory, like this:

```
$ cd /path/to/tneokernel/src
$ cp ./tn_cfg_default.h /path/to/main/project/lib_cfg/tn_cfg.h
$ ln -s /path/to/main/project/lib_cfg/tn_cfg.h ./tn_cfg.h
```

Default configuration file contains detailed comments, so you can read them and configure behavior as you like.

Definition in file tn_cfg_default.h.

**Macros**

- #define TN_CHECK_PARAM 1

    *Enables additional param checking for most of the system functions.*
- #define TN_DEBUG 0

    *Allows additional internal self-checking, useful to catch internal TNeoKernel bugs as well as illegal kernel usage (e.g.*
- #define TN_OLD_TNKERNEL_NAMES 1

    *Whether old TNKernel names (definitions, functions, etc) should be available.*
- #define TN_USE_MUTEXES 1

    *Whether mutexes API should be available.*

- #define TN_MUTEX_REC 1

  *Whether mutexes should allow recursive locking/unlocking.*

- #define TN_MUTEX_DEADLOCK_DETECT 1

  *Whether RTOS should detect deadlocks and notify user about them via callback.*

- #define TN_API_MAKE_ALIG_ARG TN_API_MAKE_ALIG_ARG__SIZE

  *API option for MAKE_ALIG() macro.*

### 14.14.2 Macro Definition Documentation

#### 14.14.2.1 #define TN_CHECK_PARAM 1

Enables additional param checking for most of the system functions.

It's surely useful for debug, but probably better to remove in release. If it is set, most of the system functions are able to return two additional codes:

- TN_RC_WPARAM if wrong params were given;

- TN_RC_INVALID_OBJ if given pointer doesn't point to a valid object. Object validity is checked by means of the special ID field of type `enum TN_ObjId`.

**See also**

    `enum TN_ObjId`

Definition at line 89 of file tn_cfg_default.h.

#### 14.14.2.2 #define TN_DEBUG 0

Allows additional internal self-checking, useful to catch internal TNeoKernel bugs as well as illegal kernel usage (e.g.

sleeping in the idle task callback). Produces a couple of extra instructions which usually just causes debugger to stop if something goes wrong.

Definition at line 99 of file tn_cfg_default.h.

#### 14.14.2.3 #define TN_OLD_TNKERNEL_NAMES 1

Whether old TNKernel names (definitions, functions, etc) should be available.

If you're porting your existing application written for TNKernel, it is definitely worth enabling. If you start new project with TNeoKernel from scratch, it's better to avoid old names.

Definition at line 109 of file tn_cfg_default.h.

#### 14.14.2.4 #define TN_MUTEX_DEADLOCK_DETECT 1

Whether RTOS should detect deadlocks and notify user about them via callback.

**See also**

    see `tn_callback_deadlock_set()`

Definition at line 133 of file tn_cfg_default.h.

**14.14.2.5 #define TN_API_MAKE_ALIG_ARG TN_API_MAKE_ALIG_ARG__SIZE**

API option for MAKE_ALIG() macro.

There is a terrible mess with MAKE_ALIG() macro: original TNKernel docs specify that the argument of it should be the size to align, but almost all ports, including "original" one, defined it so that it takes type, not size.

But the port by AlexB implemented it differently (i.e. accordingly to the docs)

When I was moving from the port by AlexB to another one, do you have any idea how much time it took me to figure out why do I have rare weird bug? :)

So, available options:

- TN_API_MAKE_ALIG_ARG__TYPE: In this case, you should use macro like this: TN_MAKE_ALI←
  G(struct my_struct) This way is used in the majority of TNKernel ports. (actually, in all ports except the one by AlexB)

- TN_API_MAKE_ALIG_ARG__SIZE: In this case, you should use macro like this: TN_MAKE_ALI←
  G(sizeof(struct my_struct)) This way is stated in TNKernel docs and used in the port for dsPI←
  C/PIC24/PIC32 by AlexB.

Definition at line 164 of file tn_cfg_default.h.

# Index