

TNeoKernel

v1.04

Generated by Doxygen 1.8.8

Tue Nov 4 2014 04:27:59



# Contents

<b>1</b>	<b>TNeoKernel overview</b>	<b>1</b>
<b>2</b>	<b>Foreword</b>	<b>3</b>
<b>3</b>	<b>Quick guide</b>	<b>5</b>
3.1	Time ticks . . . . .	5
3.2	Starting the kernel . . . . .	5
3.3	Round-robin scheduling . . . . .	9
<b>4</b>	<b>Interrupts</b>	<b>11</b>
4.1	Interrupt stack . . . . .	11
4.2	Interrupt types . . . . .	11
<b>5</b>	<b>Building the project</b>	<b>13</b>
5.1	Configuration file . . . . .	13
5.2	Building the project . . . . .	13
<b>6</b>	<b>PIC32 details</b>	<b>15</b>
6.1	Context switch . . . . .	15
6.2	Interrupts . . . . .	15
6.3	Building . . . . .	16
<b>7</b>	<b>PIC24/dsPIC details</b>	<b>17</b>
7.1	Context switch . . . . .	17
7.2	Interrupts . . . . .	17
7.3	Atomic access to the structure bit field . . . . .	18
7.4	Building . . . . .	18
<b>8</b>	<b>Why reimplement TNKernel</b>	<b>19</b>
8.1	Essential problems of TNKernel . . . . .	19
8.2	Examples of poor implementation . . . . .	19
8.2.1	One entry point, one exit point . . . . .	19
8.2.2	Don't repeat yourself . . . . .	21
8.2.3	Macros that return from function . . . . .	21

8.2.4	Code for doubly-linked lists . . . . .	22
8.3	Bugs of TNKernel 2.7 . . . . .	22
<b>9</b>	<b>Differences from TNKernel API</b>	<b>25</b>
9.1	Incompatible API changes . . . . .	25
9.1.1	System startup . . . . .	25
9.1.2	Task creation API . . . . .	25
9.1.3	Task wakeup count, activate count, suspend count . . . . .	25
9.1.4	Fixed memory pool: non-aligned address or block size . . . . .	26
9.1.5	Task service return values cleaned . . . . .	26
9.1.6	Force task releasing from wait . . . . .	27
9.1.7	Return code of tn_task_sleep() . . . . .	27
9.1.8	Events API is changed almost completely . . . . .	27
9.1.9	Zero timeout given to system functions . . . . .	27
9.2	New features . . . . .	27
9.2.1	Event group connection . . . . .	27
9.2.2	Timers . . . . .	28
9.2.3	Recursive mutexes . . . . .	28
9.2.4	Mutex deadlock detection . . . . .	28
9.2.5	New system services added . . . . .	28
9.3	Compatible API changes . . . . .	29
9.3.1	Macro MAKE_ALIG() . . . . .	29
9.3.2	Convenience macros for stack arrays definition . . . . .	29
9.3.3	Convenience macros for fixed memory block pool buffers definition . . . . .	29
9.3.4	Things renamed . . . . .	29
9.3.5	We should wait for semaphore, not acquire it . . . . .	30
9.4	Changes that do not affect API directly . . . . .	30
9.4.1	No timer task . . . . .	30
<b>10</b>	<b>Unit tests</b>	<b>33</b>
10.1	Tested CPUs . . . . .	33
10.2	How tests are implemented . . . . .	33
10.3	Get unit-tests . . . . .	35
<b>11</b>	<b>Plans</b>	<b>37</b>
11.1	Event connecting . . . . .	37
<b>12</b>	<b>Changelog</b>	<b>39</b>
12.1	v1.04 . . . . .	39
12.2	v1.03 . . . . .	39
12.3	v1.02 . . . . .	40

---

12.4 v1.01 . . . . .	40
12.5 v1.0 . . . . .	40
<b>13 Thanks</b>	<b>41</b>
<b>14 Legend</b>	<b>43</b>
<b>15 Data Structure Index</b>	<b>45</b>
15.1 Data Structures . . . . .	45
<b>16 File Index</b>	<b>47</b>
16.1 File List . . . . .	47
<b>17 Data Structure Documentation</b>	<b>49</b>
17.1 TN_DQueue Struct Reference . . . . .	49
17.1.1 Detailed Description . . . . .	49
17.2 TN_DQueueTaskWait Struct Reference . . . . .	50
17.2.1 Detailed Description . . . . .	50
17.3 TN_EGrpLink Struct Reference . . . . .	50
17.3.1 Detailed Description . . . . .	50
17.4 TN_EGrpTaskWait Struct Reference . . . . .	50
17.4.1 Detailed Description . . . . .	50
17.5 TN_EventGrp Struct Reference . . . . .	51
17.5.1 Detailed Description . . . . .	51
17.6 TN_FMem Struct Reference . . . . .	51
17.6.1 Detailed Description . . . . .	51
17.6.2 Field Documentation . . . . .	52
17.6.2.1 block_size . . . . .	52
17.6.2.2 start_addr . . . . .	52
17.7 TN_FMemTaskWait Struct Reference . . . . .	52
17.7.1 Detailed Description . . . . .	52
17.8 TN_ListItem Struct Reference . . . . .	52
17.8.1 Detailed Description . . . . .	52
17.9 TN_Mutex Struct Reference . . . . .	53
17.9.1 Detailed Description . . . . .	53
17.10 TN_Sem Struct Reference . . . . .	53
17.10.1 Detailed Description . . . . .	53
17.11 TN_Task Struct Reference . . . . .	54
17.11.1 Detailed Description . . . . .	54
17.11.2 Field Documentation . . . . .	55
17.11.2.1 stack_top . . . . .	55
17.11.2.2 deadlock_list . . . . .	55

---

17.11.2.3 subsys_wait . . . . .	55
17.11.2.4 priority_already_updated . . . . .	56
17.11.2.5 waited . . . . .	56
17.12 TN_Timer Struct Reference . . . . .	56
17.12.1 Detailed Description . . . . .	56
<b>18 File Documentation</b> . . . . .	<b>57</b>
18.1 arch/example/tn_arch_example.h File Reference . . . . .	57
18.1.1 Detailed Description . . . . .	57
18.1.2 Macro Definition Documentation . . . . .	58
18.1.2.1 _TN_FFS . . . . .	58
18.1.2.2 _TN_FATAL_ERROR . . . . .	58
18.1.2.3 TN_ARCH_STK_ATTR_BEFORE . . . . .	58
18.1.2.4 TN_ARCH_STK_ATTR_AFTER . . . . .	59
18.1.2.5 TN_PRIORITIES_MAX_CNT . . . . .	59
18.1.2.6 TN_INTSAVE_DATA . . . . .	59
18.1.2.7 TN_INTSAVE_DATA_INT . . . . .	59
18.1.2.8 TN_INT_DIS_SAVE . . . . .	59
18.1.2.9 TN_INT_RESTORE . . . . .	60
18.1.2.10 TN_INT_IDIS_SAVE . . . . .	60
18.1.2.11 TN_INT_IRESTORE . . . . .	60
18.1.2.12 _TN_SIZE_BYTES_TO_UWORDS . . . . .	60
18.1.3 Typedef Documentation . . . . .	60
18.1.3.1 TN_UWord . . . . .	60
18.1.3.2 TN_UIntPtr . . . . .	61
18.2 arch/pic24_dspic/tn_arch_pic24.h File Reference . . . . .	61
18.2.1 Detailed Description . . . . .	61
18.2.2 Macro Definition Documentation . . . . .	61
18.2.2.1 tn_p24_soft_isr . . . . .	61
18.3 arch/pic24_dspic/tn_arch_pic24_bfa.h File Reference . . . . .	61
18.3.1 Detailed Description . . . . .	61
18.3.2 Macro Definition Documentation . . . . .	62
18.3.2.1 TN_BFA_SET . . . . .	62
18.3.2.2 TN_BFA_CLR . . . . .	62
18.3.2.3 TN_BFA_INV . . . . .	62
18.3.2.4 TN_BFA_WR . . . . .	62
18.3.2.5 TN_BFA_RD . . . . .	62
18.3.2.6 TN_BFA . . . . .	63
18.3.2.7 TN_BFAR . . . . .	63
18.4 arch/pic32/tn_arch_pic32.h File Reference . . . . .	64

18.4.1	Detailed Description	64
18.4.2	Macro Definition Documentation	65
18.4.2.1	tn_p32_soft_isr	65
18.4.2.2	tn_p32_srs_isr	65
18.4.3	Variable Documentation	65
18.4.3.1	tn_p32_int_nest_count	65
18.4.3.2	tn_p32_user_sp	65
18.4.3.3	tn_p32_int_sp	66
18.5	arch/tn_arch.h File Reference	66
18.5.1	Detailed Description	66
18.5.2	Function Documentation	66
18.5.2.1	tn_arch_int_dis	66
18.5.2.2	tn_arch_int_en	66
18.5.2.3	tn_arch_sr_save_int_dis	67
18.5.2.4	tn_arch_sr_restore	67
18.5.2.5	_tn_arch_sys_init	67
18.5.2.6	_tn_arch_stack_top_get	67
18.5.2.7	_tn_arch_stack_init	67
18.5.2.8	_tn_arch_inside_isr	68
18.5.2.9	_tn_arch_is_int_disabled	68
18.5.2.10	_tn_arch_context_switch_pend	68
18.5.2.11	_tn_arch_context_switch_now_nosave	69
18.6	core/tn_cfg_dispatch.h File Reference	69
18.6.1	Detailed Description	69
18.6.2	Macro Definition Documentation	69
18.6.2.1	TN_API_MAKE_ALIG_ARG__TYPE	69
18.6.2.2	TN_API_MAKE_ALIG_ARG__SIZE	70
18.7	core/tn_common.h File Reference	70
18.7.1	Detailed Description	70
18.7.2	Macro Definition Documentation	71
18.7.2.1	TN_MAKE_ALIG_SIZE	71
18.7.2.2	TN_MAKE_ALIG	71
18.7.3	Typedef Documentation	71
18.7.3.1	TN_Timeout	71
18.7.4	Enumeration Type Documentation	72
18.7.4.1	TN_ObjId	72
18.7.4.2	TN_RCode	72
18.8	core/tn_dqueue.h File Reference	73
18.8.1	Detailed Description	73
18.8.2	Function Documentation	74

18.8.2.1	tn_queue_create	74
18.8.2.2	tn_queue_delete	75
18.8.2.3	tn_queue_send	75
18.8.2.4	tn_queue_send_polling	75
18.8.2.5	tn_queue_isend_polling	76
18.8.2.6	tn_queue_receive	76
18.8.2.7	tn_queue_receive_polling	76
18.8.2.8	tn_queue_ireceive_polling	76
18.8.2.9	tn_queue_eventgrp_connect	76
18.8.2.10	tn_queue_eventgrp_disconnect	77
18.9	core/tn_eventgrp.h File Reference	77
18.9.1	Detailed Description	77
18.9.2	Connecting an event group to other system objects	77
18.9.3	Enumeration Type Documentation	79
18.9.3.1	TN_EGrpWaitMode	79
18.9.3.2	TN_EGrpOp	79
18.9.4	Function Documentation	79
18.9.4.1	tn_eventgrp_create	79
18.9.4.2	tn_eventgrp_delete	80
18.9.4.3	tn_eventgrp_wait	80
18.9.4.4	tn_eventgrp_wait_polling	80
18.9.4.5	tn_eventgrp_await_polling	81
18.9.4.6	tn_eventgrp_modify	81
18.9.4.7	tn_eventgrp_imodify	81
18.10	core/tn_fmем.h File Reference	81
18.10.1	Detailed Description	81
18.10.2	Macro Definition Documentation	82
18.10.2.1	TN_FMЕМ_BUF_DEF	82
18.10.3	Function Documentation	83
18.10.3.1	tn_fmем_create	83
18.10.3.2	tn_fmем_delete	83
18.10.3.3	tn_fmем_get	84
18.10.3.4	tn_fmем_get_polling	84
18.10.3.5	tn_fmем_iget_polling	84
18.10.3.6	tn_fmем_release	84
18.10.3.7	tn_fmем_irelease	85
18.11	core/tn_list.h File Reference	85
18.11.1	Detailed Description	85
18.12	core/tn_mutex.h File Reference	85
18.12.1	Detailed Description	85



18.12.2 Enumeration Type Documentation	86
18.12.2.1 TN_MutexProtocol	86
18.12.3 Function Documentation	87
18.12.3.1 tn_mutex_create	87
18.12.3.2 tn_mutex_delete	87
18.12.3.3 tn_mutex_lock	87
18.12.3.4 tn_mutex_lock_polling	88
18.12.3.5 tn_mutex_unlock	88
18.13core/tn_oldsymbols.h File Reference	89
18.13.1 Detailed Description	89
18.13.2 Macro Definition Documentation	92
18.13.2.1 MAKE_ALIG	92
18.14core/tn_sem.h File Reference	92
18.14.1 Detailed Description	92
18.14.2 Function Documentation	93
18.14.2.1 tn_sem_create	93
18.14.2.2 tn_sem_delete	93
18.14.2.3 tn_sem_signal	93
18.14.2.4 tn_sem_ismal	94
18.14.2.5 tn_sem_wait	94
18.14.2.6 tn_sem_wait_polling	94
18.14.2.7 tn_sem_await_polling	94
18.15core/tn_sys.h File Reference	95
18.15.1 Detailed Description	95
18.15.2 Macro Definition Documentation	96
18.15.2.1 TN_STACK_ARR_DEF	96
18.15.3 Typedef Documentation	96
18.15.3.1 TN_CBUUserTaskCreate	96
18.15.3.2 TN_CBIdle	97
18.15.3.3 TN_CBDeadlock	97
18.15.4 Enumeration Type Documentation	97
18.15.4.1 TN_StateFlag	97
18.15.4.2 TN_Context	98
18.15.5 Function Documentation	98
18.15.5.1 tn_sys_start	98
18.15.5.2 tn_tick_int_processing	98
18.15.5.3 tn_sys_tslice_set	99
18.15.5.4 tn_sys_time_get	99
18.15.5.5 tn_callback_deadlock_set	99
18.15.5.6 tn_sys_state_flags_get	99

18.15.5.7 tn_sys_context_get . . . . .	99
18.15.5.8 tn_is_task_context . . . . .	100
18.15.5.9 tn_is_isr_context . . . . .	100
18.15.5.10 tn_cur_task_get . . . . .	100
18.15.5.11 tn_cur_task_body_get . . . . .	100
18.16 core/tn_tasks.h File Reference . . . . .	101
18.16.1 Detailed Description . . . . .	101
18.16.2 Task . . . . .	101
18.16.3 Task states . . . . .	101
18.16.4 Creating/starting tasks . . . . .	101
18.16.5 Stopping/deleting tasks . . . . .	101
18.16.6 Scheduling rules . . . . .	101
18.16.7 Idle task . . . . .	102
18.16.8 Enumeration Type Documentation . . . . .	103
18.16.8.1 TN_TaskState . . . . .	103
18.16.8.2 TN_WaitReason . . . . .	103
18.16.8.3 TN_TaskCreateOpt . . . . .	104
18.16.8.4 TN_TaskExitOpt . . . . .	104
18.16.9 Function Documentation . . . . .	105
18.16.9.1 tn_task_create . . . . .	105
18.16.9.2 tn_task_suspend . . . . .	106
18.16.9.3 tn_task_resume . . . . .	106
18.16.9.4 tn_task_sleep . . . . .	107
18.16.9.5 tn_task_wakeup . . . . .	107
18.16.9.6 tn_task_iwakeup . . . . .	107
18.16.9.7 tn_task_activate . . . . .	108
18.16.9.8 tn_task_iactivate . . . . .	108
18.16.9.9 tn_task_release_wait . . . . .	108
18.16.9.10 tn_task_irelease_wait . . . . .	109
18.16.9.11 tn_task_exit . . . . .	109
18.16.9.12 tn_task_terminate . . . . .	109
18.16.9.13 tn_task_delete . . . . .	110
18.16.9.14 tn_task_state_get . . . . .	110
18.16.9.15 tn_task_change_priority . . . . .	110
18.17 core/tn_timer.h File Reference . . . . .	111
18.17.1 Detailed Description . . . . .	111
18.17.2 Implementation of timers . . . . .	111
18.17.3 Typedef Documentation . . . . .	113
18.17.3.1 TN_TimerFunc . . . . .	113
18.17.4 Function Documentation . . . . .	113

---

18.17.4.1 tn_timer_create	113
18.17.4.2 tn_timer_delete	114
18.17.4.3 tn_timer_start	114
18.17.4.4 tn_timer_cancel	115
18.17.4.5 tn_timer_set_func	116
18.17.4.6 tn_timer_is_active	116
18.17.4.7 tn_timer_time_left	116
18.18tn.h File Reference	117
18.18.1 Detailed Description	117
18.19tn_cfg_default.h File Reference	117
18.19.1 Detailed Description	117
18.19.2 Macro Definition Documentation	118
18.19.2.1 TN_PRIORITIES_CNT	118
18.19.2.2 TN_CHECK_PARAM	118
18.19.2.3 TN_DEBUG	118
18.19.2.4 TN_OLD_TNKERNEL_NAMES	119
18.19.2.5 TN_MUTEX_DEADLOCK_DETECT	119
18.19.2.6 TN_TICK_LISTS_CNT	119
18.19.2.7 TN_API_MAKE_ALIG_ARG	119
18.19.2.8 TN_P24_SYS_IPL	119
18.19.2.9 TN_P24_SYS_IPL_STR	120
<b>Index</b>	<b>121</b>



# 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/PIC24/dsPIC, and it will be ported at least to ARM Cortex M3.

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](#)
- [Interrupts](#)
- [Building the project](#)
- [PIC32 details](#)
- [PIC24/dsPIC details](#)
- [Why reimplement TNKernel](#)
- [Differences from TNKernel API](#)
- [Unit tests](#)
- [Plans](#)
- [Changelog](#)
- [Thanks](#)
- [Legend](#)

API reference:

- [System services](#)
- [Tasks](#)
- [Mutexes](#)
- [Semaphores](#)

- Fixed-memory blocks pool
- Event groups
- Data queues
- Timers

## 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 TNKernel 2.7](#). Unit tests are, of 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 TNKernel 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.

For the full thanks list, refer to the page [Thanks](#).





# 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_p32_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_STACK_ARR_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_CBUUserTaskCreate()` callback, in which your initial task is created with `TN_TASK_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/basic/arch/pic32` directory.

### Attention

Before trying to build examples, please read [Building the project](#) page carefully: you need to copy configuration 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`.

```
/**
 * 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_STACK_ARR_DEF() for that.
TN_STACK_ARR_DEF(idle_task_stack, IDLE_TASK_STACK_SIZE);
TN_STACK_ARR_DEF(interrupt_stack, INTERRUPT_STACK_SIZE);

TN_STACK_ARR_DEF(task_a_stack, TASK_A_STK_SIZE);
TN_STACK_ARR_DEF(task_b_stack, TASK_B_STK_SIZE);
TN_STACK_ARR_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_p32_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);

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

/**
 * Application init: called from the first created application task
 */
void appl_init(void)
{
    //-- configure LED port pins
    mPORTESetPinsDigitalOut(BIT_0 | BIT_1 | BIT_2);
    mPORTEClearBits(BIT_0 | BIT_1 | BIT_2);

    //-- 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)
{
#ifdef 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

# Interrupts

### 4.1 Interrupt stack

TNeoKernel provides a separate stack for interrupt handlers. This approach could save a lot of RAM: interrupt can happen at any moment of time, and if there's no separate interrupt stack, then each task should have enough stack space for the worse case of interrupt nesting.

Assume application's ISRs take max 64 words ( $64 * 4 = 256$  bytes on PIC32) and application has 4 tasks (plus one idle task). Then, each of 5 tasks must have 64 words for interrupts:  $64 * 5 * 4 = 1280$  bytes of RAM just for 64 words for ISR.

With separate stack for interrupts, these 64 words should be allocated just once. Interrupt stack array should be given to `tn_sys_start()`. For additional information, refer to the section [Starting the kernel](#).

In order to make particular ISR use separate interrupt stack, this ISR should be defined by kernel-provided macro, which is platform-dependent: see:

- [PIC32 interrupts](#),
- [PIC24/dsPIC interrupts](#).

### 4.2 Interrupt types

On some platforms (namely, on PIC24/dsPIC), there are two types of interrupts: *system interrupts* and *user interrupts*. Other platforms have *system interrupts* only. Kernel services are allowed to call only from *system interrupts*, and interrupt-related kernel services (`tn_arch_sr_save_int_dis()`, `tn_arch_sr_restore()`, `tn_arch_inside_isr()`, etc) affect **only system interrupts**. Say, if `tn_arch_inside_isr()` is called from *user interrupt*, it returns 0.

Particular platform might have additional constraints for each of these interrupt types, refer to the details of each supported platform for details.





## Chapter 5

# Building the project

Some notes on building the project

### 5.1 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.

### 5.2 Building the project

For information on building TNeoKernel, please refer to the appropriate section for your platform:

- [Building for PIC24/dsPIC](#)
- [Building for PIC32](#)



## Chapter 6

# PIC32 details

PIC32 port implementation details

### 6.1 Context switch

The context switch is implemented using the core software 0 interrupt (CS0), which is configured by the kernel to the lowest priority (1). This interrupt is handled completely by the kernel, application should never touch it.

The interrupt priority level 1 should not be configured to use shadow register sets.

Multi-vectored interrupt mode should be enabled.

#### 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. Do note that if we include this file in the TNeoKernel library project, it doesn't work for vector, unfortunately.

If you forgot to include this file, you got an error on the link step, like this:

```
undefined reference to `_you_should_add_file__tn_arch_pic32_int_vec1_S__to_the_project'
```

Which is much more informative than if you just get to `_DefaultInterrupt` when it's time to switch context.

### 6.2 Interrupts

For generic information about interrupts in TNeoKernel, refer to the page [Interrupts](#).

PIC32 port has *system interrupts* only, there are no *user 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_p32_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_p32_srs_isr(_UART_1_VECTOR)  
{  
    /* here is your ISR code, including clearing of interrupt flag, and so on */  
}
```

In spite of the fact that the kernel provides separate stack for interrupt, this isn't a mandatory on PIC32: you're able to define your ISR in a standard way, making it use stack of interrupted task and work a bit faster. Like this:

```
void __ISR(_TIMER_1_VECTOR) timer_1_isr(void)
{
    /* here is your ISR code, including clearing of interrupt flag, and so on */
}
```

There is always a tradeoff. There are **no additional constraints** on ISR defined without kernel-provided macro: in either ISR, you can call the same set of kernel services.

When you make a decision on whether particular ISR should use separate stack, consider the following:

- When ISR is defined in a standard way, and no function is called from that ISR, only necessary registers are saved on stack. If you have such an ISR (that doesn't call any function), and this ISR should work very fast, consider using standard way instead of kernel-provided macro.
- When ISR is defined in a standard way, but it calls any function and doesn't use shadow register set, compiler saves (almost) full context **on the task's stack**, because it doesn't know which registers are used inside the function. In this case, it usually makes more sense to use kernel-provided macro (see below).
- Kernel-provided interrupt macros switch stack pointer between interrupt stack and task stack automatically, it takes additional time: e.g. on PIC32 it's about 20 cycles.
- Kernel-provided interrupt macro that doesn't use shadow register set always saves (almost) full context **on the interrupt stack**, independently of whether any function is called from an ISR.
- Kernel-provided interrupt macro that uses shadow register set saves a little amount of registers **on the interrupt stack**.

## 6.3 Building

For generic information on building TNeoKernel, refer to the page [Building the project](#).

MPLABX project for PIC32 port resides in the `src/arch/pic32/tneokernel_pic32.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_pic32.X.a` file in whatever way you like.

# Chapter 7

## PIC24/dsPIC details

PIC24/dsPIC port implementation details

### 7.1 Context switch

The context switch is implemented using the external interrupt 0 (`INT0`). It is handled completely by the kernel, application should never touch it.

### 7.2 Interrupts

For detailed information about interrupts in TNeoKernel, refer to the page [Interrupts](#).

PIC24/dsPIC TNeoKernel port supports nested interrupts. It allows to specify the range of *system interrupt priorities*. Refer to the section [Interrupt types](#) for details on what is *system interrupt*.

System interrupts use separate interrupt stack instead of the task's stack. This approach saves a lot of RAM.

The range is specified by just a single number: `TN_P24_SYS_IPL`, which represents maximum *system interrupt priority*. Here is a list of available priorities and their characteristics:

- priorities [1 .. `TN_P24_SYS_IPL`]:
  - Kernel services **are** allowed to call;
  - The macro `tn_p24_soft_isr()` **must** be used.
  - Separate interrupt stack is used by ISR;
  - Interrupts get disabled for short periods of time when modifying critical kernel data (for about 100 cycles or the like).
- priorities [(`TN_P24_SYS_IPL` + 1) .. 6]:
  - Kernel services **are not** allowed to call;
  - The macro `tn_p24_soft_isr()` **must not** be used.
  - Task's stack is used by ISR;
  - Interrupts are not disabled when modifying critical kernel data, but they are disabled for 4..8 cycles by `disi` instruction when entering/exiting system ISR: we need to safely modify `SP` and `SPLIM`.
- priority 7:
  - Kernel services **are not** allowed to call;
  - The macro `tn_p24_soft_isr()` **must not** be used.
  - Task's stack is used by ISR;

- Interrupts are never disabled by the kernel. Note that `disi` instruction leaves interrupts of priority 7 enabled.

The kernel provides C-language macro for calling C-language **system** interrupt service routines.

Usage is as follows:

```
/*
 * Timer 1 interrupt handler using software interrupt context saving,
 * PSV is handled automatically:
 */
tn_p24_soft_isr(_T1Interrupt, auto_psv)
{
    //-- clear interrupt flag
    IFS0bits.T1IF = 0;

    //-- do something useful
}
```

#### Attention

do **not** use this macro for non-system interrupt (that is, for interrupt of priority higher than [TN\\_P24\\_SYS\\_I↔PI](#)). Use standard way to define it. If you violate this rule, debugger will be halted by the kernel when entering ISR. In release build, CPU is just reset.

## 7.3 Atomic access to the structure bit field

The problem with PIC24/dsPIC is that when we write something like:

```
IPC0bits.INT0IP = 0x05;
```

We actually have read-modify-write sequence which can be interrupted, so that resulting data could be corrupted. PIC24/dsPIC port provides several macros that offer atomic access to the structure bit field.

The kernel would not probably provide that kind of functionality, but TNeoKernel itself needs it, so, it is made public so that application can use it too.

Refer to the page [Atomic bit-field access macros](#) for details.

## 7.4 Building

For generic information on building TNeoKernel, refer to the page [Building the project](#).

MPLABX project for PIC24/dsPIC port resides in the `src/arch/pic24_dspic/tneokernel_pic24_↔dspic.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` ....

#### Attention

there are two configurations of this project: `eds` and `no_eds`, for devices with and without extended data space, respectively. When you add library project to your application project, you should select correct configuration for your device; otherwise, you get "undefined reference" errors at linker step.

Alternatively, of course you can just build it and use resulting `.a` file in whatever way you like.

## Chapter 8

# Why reimplement TNKernel

Explanation of essential TNKernel problems as well as several examples of poor implementation.

### 8.1 Essential problems of TNKernel

- The most essential problem is that TNKernel is a very hastily-written project. Several concepts are just poorly thought out, others are poorly implemented: there is a lot of code duplication and inconsistency;
- It is untested: there are no unit tests for the kernel, this is not acceptable for the project like real-time kernel;

As a result of the two above, the kernel is buggy. And even more, the kernel is really **hard to maintain** because of inconsistency, so when we add new features or change something, we are likely to add new bugs as well.

- It is unsupported. I've written to the Yuri Tiomkin about [troubles with MAKE\\_ALIG\(\) macro](#) as well as about bugs in the kernel, my messages were just ignored;
- Documentation is far from perfect and it lives separately of the project itself: latest kernel version at the moment is 2.7 (published at 2013), but latest documentation is for 2.3 (published at 2006).

### 8.2 Examples of poor implementation

#### 8.2.1 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;
}
```

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 rc = SUCCESS;
    tn_disable_interrupt();
    //-- do something

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

out:
    tn_enable_interrupt();
    return rc;
}

```

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, let's go further:

While multiple `goto`-s to single label are better than multiple `return` statements, it becomes less useful as we get to something more complicated. Imagine we need to perform some checks *before* disabling interrupts, and perform some other checks *after* disabling them. Then, we have to create two labels, like that:

```

int my_function(void)
{
    int rc = SUCCESS;

    if (error1()){
        rc = ERROR1;
        goto out;
    }

    tn_disable_interrupt();

    if (error2()){
        rc = ERROR2;
        goto out_ei;
    }

    if (error3()){
        rc = ERROR3;
        goto out_ei;
    }

    //-- perform job

out_ei:
    tn_enable_interrupt();

out:
    return rc;
}

```

For each error handling, we should specify the label explicitly, and it's easy to mix labels up, especially if we add some new case to check in the future. So, I believe this approach is a superior:

```

int my_function(void)
{
    int rc = SUCCESS;

    if (error1()){
        rc = ERROR1;
    } else {
        tn_disable_interrupt();

        if (error2()){
            rc = ERROR2;
        } else if (error3()){
            rc = ERROR3;
        } else {
            //-- perform job
        }

        tn_enable_interrupt();
    }

    return rc;
}

```



Then, for each new error handling, we should just add new `else if` block, and there's no need to care where to go if error happened. Let the compiler do the branching job for you. More, this code looks more compact.

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.

### 8.2.2 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 setting 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](#)).

### 8.2.3 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
}
```

## 8.2.4 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_queue(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`.

## 8.3 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`. The same for queue deletion, semaphore deletion, event deletion.
- `tn_sys_tslice_ticks()` : if wrong params are given, `TERR_WRONG_PARAM` is returned and interrupts remain disabled.
- `tn_queue_receive()` and `tn_fmем_get()` : if timeout is in effect, then `TN_RC_TIMEOUT` is returned, but user-provided pointer is altered anyway (some garbage data is written there)
- 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
- Event: if `TN_EVENT_ATTR_CLR` flag is set, and the task that is waiting for event is suspended, this flag `TN_EVENT_ATTR_CLR` is ignored (pattern is not reset). I can't say this bug is "fixed" because TNeKernel has [event groups instead of events](#), and there is no `TN_EVENT_ATTR_CLR` flag.

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



## Chapter 9

# Differences from TNKernel API

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

### 9.1 Incompatible API changes

#### 9.1.1 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.

#### 9.1.2 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.

#### 9.1.3 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.

### 9.1.4 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 particular 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.

TNeoKernel also provides convenience macro `TN_FMEM_BUF_DEF()` for buffer definition, so, as a generic rule, it is good practice to define buffers for memory pool like this:

```
//-- 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_fmем_buf, struct MyMemoryItem, MY_MEMORY_BUF_SIZE);

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

And then, construct your `my_fmем` as follows:

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

### 9.1.5 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.

### 9.1.6 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.

### 9.1.7 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()`;

### 9.1.8 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.

TNeoKernel also offers a very useful feature: connecting an event group to other kernel objects. Read [below](#).

For detailed API reference, refer to the [tn\\_eventgrp.h](#).

### 9.1.9 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.

## 9.2 New features

### 9.2.1 Event group connection

Sometimes task needs to wait for different system events, the most common examples are:

- wait for a message from the queue(s) plus wait for some application-dependent event (such as a flag to finish the task, or whatever);
- wait for messages from multiple queues.

If the kernel doesn't offer a mechanism for that, programmer usually have to use polling services on these queues and sleep for a few system ticks. Obviously, this approach has serious drawbacks: we have a lot of useless context switches, and response for the message gets much slower. Actually, we lost the main goal of the preemptive kernel when we use polling services like that.

TNeoKernel (since [v1.03](#)) offers a solution: an event group can be connected to other kernel objects, and these objects will maintain certain flags inside that event group automatically.

Refer to the section [Connecting an event group to other system objects](#) for details.

### 9.2.2 Timers

Support of timers was added since TNeoKernel [v1.02](#).

Timer is a kernel object that is used to ask the kernel to call some user-provided function at a particular time in the future, based on the [system timer](#) tick.

If you need to repeatedly wake up particular task, you can create semaphore which you should [wait for](#) in the task, and [signal](#) in the timer callback.

If you need to perform rather fast action (such as toggle some pin, or the like), consider doing that right in the timer callback, in order to avoid context switch overhead.

The timer callback approach provides ultimate flexibility.

For details, refer to the [timers documentation](#).

### 9.2.3 Recursive mutexes

Sometimes I feel lack of mutexes that allow recursive locking. 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.

### 9.2.4 Mutex deadlock detection

TNeoKernel can notify you by calling your callback function if deadlock occurs. See:

- compile-time option [TN\\_MUTEX\\_DEADLOCK\\_DETECT](#),
- [tn\\_callback\\_deadlock\\_set\(\)](#),
- [TN\\_CBDeadlock](#).

### 9.2.5 New system services added

Several system services were added:

- [tn\\_cur\\_task\\_get\(\)](#) : return a pointer to the `struct TN_Task` of the currently running task;
- [tn\\_cur\\_task\\_body\\_get\(\)](#) : return pointer to the currently running task body function;
- [tn\\_task\\_state\\_get\(\)](#) : get state of the task.



## 9.3 Compatible API changes

### 9.3.1 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_ALIG_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.

### 9.3.2 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_STACK_ARR_DEF()`. See `tn_task_create()` for the usage example.

### 9.3.3 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_fmем_create()` for usage example.

### 9.3.4 Things renamed

There is a lot of inconsistency with naming stuff in original TNKernel:

- Why do we have `tn_queue_send_polling()` / `tn_queue_isend_polling()` (notice the `i` letter before the verb, not before `polling`), but `tn_fmем_get_polling()` / `tn_fmем_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.

### 9.3.5 We should wait for semaphore, not acquire it

One of the renamings deserves special mentioning: `tn_sem_acquire()` and friends are renamed to `tn_↔sem_wait()` and friends. That's because names acquire/release are actually misleading for the semaphore: semaphore is a *signaling mechanism*, and **not** the locking mechanism.

Actually, there's a lot of confusion about usage of mutexes/semaphores, so it's quite recommended to read small article by Michael Barr: [Mutexes and Semaphores Demystified](#).

Old names (`tn_sem_acquire()` and friends) are still available through `tn_oldsymbols.h`.

## 9.4 Changes that do not affect API directly

### 9.4.1 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 10

## Unit tests

Brief information on the implementation of unit tests

### 10.1 Tested CPUs

Currently, unit tests project is tested in the hardware on the following CPUs:

- PIC32MX440F512H
- PIC24FJ256GB106

### 10.2 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.

## 10.3 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](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](http://hg.dfrank.ru/tntest/project_common)





# Chapter 11

## Plans

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

### 11.1 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()`. When event happened, we just check which flags are set, and get message from appropriate queue.



# Chapter 12

## Changelog

TNeoKernel changelog

### 12.1 v1.04

Release date: 2014-11-04.

- Added PIC24/dsPIC support, refer to the page [PIC24/dsPIC details](#);
- PIC32: Core Software Interrupt is now handled by the kernel completely, application shouldn't set it up anymore. Refer to the page [PIC32 details](#).
- Refactor: the following symbols: `NULL`, `BOOL`, `TRUE`, `FALSE` now have the `TN_` prefix: `TN_NULL`, `TN_BOOL`, `TN_TRUE`, `TN_FALSE`. This is because non-prefixed symbols may be defined by some other program module, which leads to conflicts. The easiest and robust way is to add unique prefix.
- Refactor: PIC32 MPLABX project renamed from `tneokernel.X` to `tneokernel_pic32.X`.
- Refactor: PIC32 ISR macros renamed: `tn_soft_isr()` -> `tn_p32_soft_isr()`, `tn_srs_isr()` -> `tn_p32_srs_isr()`. It is much easier to maintain documentation for different macros if they have different names; more, the signature of these macros is architecture-dependent. Old names are also available for backward compatibility.

### 12.2 v1.03

Release date: 2014-10-20.

- Added a capability to connect an [event group](#) to other system objects, particularly to the [queue](#). This offers a way to wait for messages from multiple queues with just a single system call. Refer to the section [Connecting an event group to other system objects](#) for details. Example project that demonstrates that technique is also available: `examples/queue_eventgrp_conn`.
- PIC32 Interrupts: this isn't a mandatory anymore to use kernel-provided macros `tn_p32_soft_isr()` or `tn_p32_srs_isr()`: interrupts can be defined with standard way too: this particular ISR will use task's stack instead of interrupt stack, therefore it takes much more RAM and works a bit faster. There are no additional constraints on ISR defined without kernel-provided macro: in either ISR, you can call the same set of kernel services. Refer to the page [Interrupts](#) for details.
- Priority 0 is now allowed to use by application (in the original TNKernel, it was reserved for the timer task, but TNeoKernel [does not have timer task](#))

- Application is now available to specify how many priority levels does it need for, it helps to save a bit of RAM. For details, refer to [TN\\_PRIORITIES\\_CNT](#).
- Added example project `examples/queue` that demonstrates the pattern on how to use [queue](#) together with [fixed memory pool](#) effectively.

## 12.3 v1.02

Release date: 2014-10-14.

- Added [timers](#): kernel objects that are used to ask the kernel to call some user-provided function at a particular time in the future;
- Removed `tn_sys_time_set()` function, because now TNeoKernel uses internal system tick count for timers, and modifying system tick counter by user is a *really* bad idea.

## 12.4 v1.01

Release date: 2014-10-09.

- **FIX:** `tn_queue_receive()` and `tn_fmem_get()` : if non-zero `timeout` is in effect, then `TN_RC_↔TIMEOUT` is returned, but user-provided pointer is altered anyway (some garbage data is written there). This bug was inherited from TNKernel.
- Added `tn_task_state_get()`
- `tn_sem_acquire()` and friends are renamed to `tn_sem_wait()` and friends. More on this read [here](#). Old name is still available through `tn_oldsymbols.h`.

## 12.5 v1.0

Release date: 2014-10-01.

- Initial stable version of TNeoKernel. Lots of work done: thorough review and re-implementation of TNKernel 2.7, implemented detailed unit tests, and so on.

## Chapter 13

# Thanks

There are people that I would like to thank:

- **Yuri Tiomkin** - for original TNKernel. Although the implementation of TNKernel 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;
- **Anders Montonen** - for original implementation of TNKernel-PIC32 port;
- **Alex Borisov** - for TNKernel port which I was using for a long time;
- **Alexey Morozov** and **Alexey Gromov**, my chiefs in the **ORION** company, for being flexible about my time;
- **Robert White** - for nice ideas.





Thank you guys. TNeoKernel would never be what it is without you.



# Chapter 14

## 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 15

## Data Structure Index

### 15.1 Data Structures

Here are the data structures with brief descriptions:

- [TN\\_DQueue](#)  
Structure representing data queue object . . . . . 49
- [TN\\_DQueueTaskWait](#)  
DQueue-specific fields related to waiting task, to be included in struct [TN\\_Task](#) . . . . . 50
- [TN\\_EGrpLink](#)  
A link to event group: used when event group can be connected to some kernel object, such as queue . . . . . 50
- [TN\\_EGrpTaskWait](#)  
EventGrp-specific fields related to waiting task, to be included in struct [TN\\_Task](#) . . . . . 50
- [TN\\_EventGrp](#)  
Event group . . . . . 51
- [TN\\_FMem](#)  
Fixed memory blocks pool . . . . . 51
- [TN\\_FMemTaskWait](#)  
FMem-specific fields related to waiting task, to be included in struct [TN\\_Task](#) . . . . . 52
- [TN\\_ListItem](#)  
Circular doubly linked list item, for internal kernel usage . . . . . 52
- [TN\\_Mutex](#)  
Mutex . . . . . 53
- [TN\\_Sem](#)  
Semaphore . . . . . 53
- [TN\\_Task](#)  
Task . . . . . 54
- [TN\\_Timer](#)  
Timer . . . . . 56



# Chapter 16

## File Index

### 16.1 File List

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

<a href="#">tn.h</a>	The main kernel header file that should be included by user application; it merely includes subsystem-specific kernel headers . . . . .	117
<a href="#">tn_cfg_default.h</a>	TNeoKernel default configuration file, to be copied as <code>tn_cfg.h</code> . . . . .	117
<a href="#">arch/tn_arch.h</a>	Architecture-dependent routines declaration . . . . .	66
<a href="#">arch/example/tn_arch_example.h</a>	Example of architecture-dependent routines . . . . .	57
<a href="#">arch/pic24_dspic/tn_arch_pic24.h</a>	PIC24/dsPIC architecture-dependent routines . . . . .	61
<a href="#">arch/pic24_dspic/tn_arch_pic24_bfa.h</a>	Atomic bit-field access macros for PIC24/dsPIC . . . . .	61
<a href="#">arch/pic32/tn_arch_pic32.h</a>	PIC32 architecture-dependent routines . . . . .	64
<a href="#">core/tn_cfg_dispatch.h</a>	Dispatch configuration: set predefined options, include user-provided <code>cfg</code> file as well as default <code>cfg</code> file . . . . .	69
<a href="#">core/tn_common.h</a>	Definitions used through the whole kernel . . . . .	70
<a href="#">core/tn_dqueue.h</a>	A data queue is a FIFO that stores pointer (of type <code>void *</code> ) in each cell, called (in uITRON style) a data element . . . . .	73
<a href="#">core/tn_eventgrp.h</a>	Event group . . . . .	77
<a href="#">core/tn_fmем.h</a>	Fixed memory blocks pool . . . . .	81
<a href="#">core/tn_list.h</a>	Circular doubly linked list, for internal kernel usage . . . . .	85
<a href="#">core/tn_mutex.h</a>	A mutex is an object used to protect shared resource . . . . .	85
<a href="#">core/tn_oldsymbols.h</a>	Compatibility layer for old projects that use old TNKernel names; usage of them in new projects is discouraged . . . . .	89
<a href="#">core/tn_sem.h</a>	A semaphore: an object to provide signaling mechanism . . . . .	92
<a href="#">core/tn_sys.h</a>	Kernel system routines: system start, tick processing, time slice managing . . . . .	95

---

<a href="#">core/tn_tasks.h</a> . . . . .	101
<a href="#">core/tn_timer.h</a>	
Timer is a kernel object that is used to ask the kernel to call some user-provided function at a particular time in the future, based on the <a href="#">system timer</a> tick . . . . .	111

# Chapter 17

## Data Structure Documentation

### 17.1 TN\_DQueue Struct Reference

#### 17.1.1 Detailed Description

Structure representing data queue object.

Definition at line 105 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 TN\_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*
- struct [TN\\_EGrpLink](#) `eventgrp_link`  
*connected event group*

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

- [core/tn\\_dqueue.h](#)

## 17.2 TN\_DQueueTaskWait Struct Reference

### 17.2.1 Detailed Description

DQueue-specific fields related to waiting task, to be included in struct [TN\\_Task](#).

Definition at line 140 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`

## 17.3 TN\_EGrpLink Struct Reference

### 17.3.1 Detailed Description

A link to event group: used when event group can be connected to some kernel object, such as queue.

Definition at line 185 of file `tn_eventgrp.h`.

#### Data Fields

- struct [TN\\_EventGrp](#) \* [eventgrp](#)  
*event group whose event(s) should be managed by other kernel object*
- [TN\\_UWord](#) [pattern](#)  
*event pattern to manage*

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

- `core/tn_eventgrp.h`

## 17.4 TN\_EGrpTaskWait Struct Reference

### 17.4.1 Detailed Description

EventGrp-specific fields related to waiting task, to be included in struct [TN\\_Task](#).

Definition at line 169 of file `tn_eventgrp.h`.

#### Data Fields

- [TN\\_UWord](#) [wait\\_pattern](#)  
*event wait pattern*
- enum [TN\\_EGrpWaitMode](#) [wait\\_mode](#)  
*event wait mode: AND or OR*

- [TN\\_UWord actual\\_pattern](#)  
*pattern that caused task to finish waiting*

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

- [core/tn\\_eventgrp.h](#)

## 17.5 TN\_EventGrp Struct Reference

### 17.5.1 Detailed Description

Event group.

Definition at line 159 of file `tn_eventgrp.h`.

#### Data Fields

- struct [TN\\_ListItem wait\\_queue](#)  
*task wait queue*
- [TN\\_UWord 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](#)

## 17.6 TN\_FMem Struct Reference

### 17.6.1 Detailed Description

Fixed memory blocks pool.

Definition at line 78 of file `tn_fmем.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_ALIGN_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*

## 17.6.2 Field Documentation

### 17.6.2.1 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_ALIG←G_SIZE()` for that.

See also

`TN_MAKE_ALIG_SIZE()`

Definition at line 88 of file `tn_fmем.h`.

### 17.6.2.2 void\* TN\_FMem::start\_addr

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

Definition at line 98 of file `tn_fmем.h`.

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

- [core/tn\\_fmем.h](#)

## 17.7 TN\_FMemTaskWait Struct Reference

### 17.7.1 Detailed Description

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

Definition at line 112 of file `tn_fmем.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\\_fmем.h](#)

## 17.8 TN\_ListItem Struct Reference

### 17.8.1 Detailed Description

Circular doubly linked list item, for internal kernel usage.

Definition at line 63 of file `tn_list.h`.

#### Data Fields

- struct `TN_ListItem` \* `prev`  
*pointer to previous item*



- struct [TN\\_ListItem](#) \* [next](#)  
*pointer to next item*

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

- [core/tn\\_list.h](#)

## 17.9 TN\_Mutex Struct Reference

### 17.9.1 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](#)

## 17.10 TN\_Sem Struct Reference

### 17.10.1 Detailed Description

Semaphore.

Definition at line 88 of file [tn\\_sem.h](#).

## Data Fields

- struct [TN\\_ListItem](#) `wait_queue`  
*List of tasks that wait for the 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](#)

## 17.11 TN\_Task Struct Reference

### 17.11.1 Detailed Description

Task.

Definition at line 241 of file `tn_tasks.h`.

## Data Fields

- [TN\\_UWord](#) \* `stack_top`  
*pointer to task's current top of the stack; 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\\_Timer](#) `timer`  
*timer object to implement task waiting for timeout*
- 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](#) \* `base_stack_top`  
*base top of the stack for this task*
- int `stack_size`  
*size of task's stack (in `sizeof(TN_UWord)`, 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)*
- 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_fmем.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.*
- unsigned [waited](#): 1
  - Flag indicates that task waited for something This flag is set automatically in `_tn_task_set_waiting()` Must be cleared manually before calling any service that could sleep, if the caller is interested in the relevant value of this flag.*

## 17.11.2 Field Documentation

### 17.11.2.1 [TN\\_UWord\\* TN\\_Task::stack\\_top](#)

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

Definition at line 245 of file `tn_tasks.h`.

### 17.11.2.2 [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 272 of file `tn_tasks.h`.

### 17.11.2.3 [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.

#### 17.11.2.4 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 343 of file `tn_tasks.h`.

#### 17.11.2.5 unsigned TN\_Task::waited

Flag indicates that task waited for something This flag is set automatically in `_tn_task_set_waiting()` Must be cleared manually before calling any service that could sleep, if the caller is interested in the relevant value of this flag.

Definition at line 349 of file `tn_tasks.h`.

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

- [core/tn\\_tasks.h](#)

## 17.12 TN\_Timer Struct Reference

### 17.12.1 Detailed Description

Timer.

Definition at line 202 of file `tn_timer.h`.

#### Data Fields

- struct [TN\\_ListItem](#) `timer_queue`  
*A list item to be included in the [system timer](#) queue.*
- [TN\\_TimerFunc](#) \* `func`  
*Function to be called by timer.*
- void \* [p\\_user\\_data](#)  
*User data pointer that is given to user-provided `func`.*
- [TN\\_Timeout](#) `timeout_cur`  
*Current (left) timeout value.*
- enum [TN\\_ObjId](#) `id_timer`  
*id for object validity verification*

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

- [core/tn\\_timer.h](#)

# Chapter 18

## File Documentation

### 18.1 arch/example/tn\_arch\_example.h File Reference

#### 18.1.1 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_MAX_CNT TN_INT_WIDTH`  
*Maximum number of priorities available, this value usually matches `TN_INT_WIDTH`.*
- `#define TN_WAIT_INFINITE (TN_Timeout)0xFFFFFFFF`  
*Value for infinite waiting, usually matches `ULONG_MAX`, because `TN_Timeout` is declared as `unsigned long`.*
- `#define TN_FILL_STACK_VAL 0xFEEDFACE`  
*Value for initializing the unused space of 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_IRESTORE()`.*
- `#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.*
- `#define _TN_CONTEXT_SWITCH_IPEND_IF_NEEDED() _tn_context_switch_pend_if_needed()`  
*Pend context switch from interrupt.*
- `#define _TN_SIZE_BYTES_TO_UWORDS(size_in_bytes) ((size_in_bytes) >> 2)`  
*Converts size in bytes to size in `TN_UWord`.*

## Typedefs

- typedef unsigned int `TN_UWord`  
*Unsigned integer type whose size is equal to the size of CPU register.*
- typedef unsigned int `TN_UIntPtr`  
*Unsigned integer type that is able to store pointers.*

## 18.1.2 Macro Definition Documentation

### 18.1.2.1 `#define TN_FFS( x ) (32 - __builtin_clz((x) & (0 - (x))))`

FFS - find first set bit.

Used in `_find_next_task_to_run()` function. Say, for `0xa8` it should return 3.

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

Definition at line 53 of file `tn_arch_example.h`.

### 18.1.2.2 `#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 62 of file `tn_arch_example.h`.

### 18.1.2.3 `#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 76 of file `tn_arch_example.h`.

#### 18.1.2.4 #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 87 of file tn\_arch\_example.h.

#### 18.1.2.5 #define TN\_PRIORITIES\_MAX\_CNT TN\_INT\_WIDTH

Maximum number of priorities available, this value usually matches [TN\\_INT\\_WIDTH](#).

See also

[TN\\_PRIORITIES\\_CNT](#)

Definition at line 119 of file tn\_arch\_example.h.

#### 18.1.2.6 #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 142 of file tn\_arch\_example.h.

#### 18.1.2.7 #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\\_IRESTORE\(\)](#).

See also

[TN\\_INT\\_IDIS\\_SAVE\(\)](#)

[TN\\_INT\\_IRESTORE\(\)](#)

Definition at line 151 of file tn\_arch\_example.h.

#### 18.1.2.8 #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 163 of file tn\_arch\_example.h.

18.1.2.9 `#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 175 of file `tn_arch_example.h`.

18.1.2.10 `#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 184 of file `tn_arch_example.h`.

18.1.2.11 `#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 193 of file `tn_arch_example.h`.

18.1.2.12 `#define _TN_SIZE_BYTES_TO_UWORDS( size_in_bytes ) ((size_in_bytes) >> 2)`

Converts size in bytes to size in `TN_UWord`.

For 32-bit platforms, we should shift it by 2 bit to the right; for 16-bit platforms, we should shift it by 1 bit to the right.

Definition at line 212 of file `tn_arch_example.h`.

### 18.1.3 Typedef Documentation

18.1.3.1 `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 104 of file `tn_arch_example.h`.



### 18.1.3.2 typedef unsigned int TN\_UIntPtr

Unsigned integer type that is able to store pointers.

We need it because some platforms don't define `uintptr_t`. Typically it's `unsigned int`.

Definition at line 111 of file `tn_arch_example.h`.

## 18.2 arch/pic24\_dspic/tn\_arch\_pic24.h File Reference

### 18.2.1 Detailed Description

PIC24/dsPIC architecture-dependent routines.

Definition in file [tn\\_arch\\_pic24.h](#).

### Macros

- #define `tn_p24_soft_isr(_func, _psv) _tn_soft_isr_internal(_func, _psv, )`  
*ISR wrapper macro for software context saving.*

### 18.2.2 Macro Definition Documentation

#### 18.2.2.1 #define `tn_p24_soft_isr( _func, _psv ) _tn_soft_isr_internal(_func, _psv, )`

ISR wrapper macro for software context saving.

Usage looks like the following:

```
tn_p24_soft_isr(_T1Interrupt, auto_psv)
{
    //-- clear interrupt flag
    IFS0bits.T1IF = 0;

    //-- do something useful
}
```

Which should be used for system interrupts, instead of standard way:

```
void __attribute__((__interrupt__, auto_psv)) _T1Interrupt(void)
```

Where `_T1Interrupt` is the usual PIC24/dsPIC ISR name, and `auto_psv` (or `no_auto_psv`) is the usual attribute argument for interrupt.

Definition at line 450 of file `tn_arch_pic24.h`.

## 18.3 arch/pic24\_dspic/tn\_arch\_pic24\_bfa.h File Reference

### 18.3.1 Detailed Description

Atomic bit-field access macros for PIC24/dsPIC.

Initially, the code was taken from the [article by Alex Borisov \(russian\)](#), and modified a bit.

The kernel would not probably provide that kind of functionality, but the kernel itself needs it, so, it is made public so that application can use it too.

Definition in file [tn\\_arch\\_pic24\\_bfa.h](#).

## Macros

- `#define TN_BFA_SET 0x1111`  
*Command for `TN_BFA ()` macro: Set bits in the bit field by mask; . . . macro param should be set to the bit mask to set.*
- `#define TN_BFA_CLR 0x2222`  
*Command for `TN_BFA ()` macro: Clear bits in the bit field by mask; . . . macro param should be set to the bit mask to clear.*
- `#define TN_BFA_INV 0x3333`  
*Command for `TN_BFA ()` macro: Invert bits in the bit field by mask; . . . macro param should be set to the bit mask to invert.*
- `#define TN_BFA_WR 0xAAAA`  
*Command for `TN_BFA ()` macro: Write bit field; . . . macro param should be set to the value to write.*
- `#define TN_BFA_RD 0xB BBBB`  
*Command for `TN_BFA ()` macro: Read bit field; . . . macro param ignored.*
- `#define TN_BFA(comm, reg_name, field_name,...)`  
*Macro for atomic access to the structure bit field.*
- `#define TN_BFAR(comm, reg_name, lower, upper,...)`  
*Macro for atomic access to the structure bit field specified as a range.*

### 18.3.2 Macro Definition Documentation

#### 18.3.2.1 `#define TN_BFA_SET 0x1111`

Command for `TN_BFA ()` macro: Set bits in the bit field by mask; . . . macro param should be set to the bit mask to set.

Definition at line 76 of file `tn_arch_pic24_bfa.h`.

#### 18.3.2.2 `#define TN_BFA_CLR 0x2222`

Command for `TN_BFA ()` macro: Clear bits in the bit field by mask; . . . macro param should be set to the bit mask to clear.

Definition at line 80 of file `tn_arch_pic24_bfa.h`.

#### 18.3.2.3 `#define TN_BFA_INV 0x3333`

Command for `TN_BFA ()` macro: Invert bits in the bit field by mask; . . . macro param should be set to the bit mask to invert.

Definition at line 84 of file `tn_arch_pic24_bfa.h`.

#### 18.3.2.4 `#define TN_BFA_WR 0xAAAA`

Command for `TN_BFA ()` macro: Write bit field; . . . macro param should be set to the value to write.

Definition at line 88 of file `tn_arch_pic24_bfa.h`.

#### 18.3.2.5 `#define TN_BFA_RD 0xB BBBB`

Command for `TN_BFA ()` macro: Read bit field; . . . macro param ignored.

Definition at line 92 of file `tn_arch_pic24_bfa.h`.

18.3.2.6 #define TN\_BFA( *comm*, *reg\_name*, *field\_name*, ... )

Macro for atomic access to the structure bit field.

The BFA acronym means Bit Field Access.

## Parameters

<i>comm</i>	command to execute: <ul style="list-style-type: none"> <li>• <a href="#">TN_BFA_WR</a> - write bit field</li> <li>• <a href="#">TN_BFA_RD</a> - read bit field</li> <li>• <a href="#">TN_BFA_SET</a> - set bits by mask</li> <li>• <a href="#">TN_BFA_CLR</a> - clear bits by mask</li> <li>• <a href="#">TN_BFA_INV</a> - invert bits by mask</li> </ul>
<i>reg_name</i>	register name (PORTA, CMCON, ...).
<i>field_name</i>	structure field name
...	used if only <i>comm</i> != <a href="#">TN_BFA_RD</a> . Meaning depends on the <i>comm</i> , see comments for specific command: <a href="#">TN_BFA_WR</a> , etc.

Usage examples:

```
int a = 0x02;

/-- Set third bit of the INTOIP field in the IPC0 register:
// IPC0bits.INTOIP |= (1 << 2);
TN_BFA(TN_BFA_SET, IPC0, INTOIP, (1 << 2));

/-- Clear second bit of the INTOIP field in the IPC0 register:
// IPC0bits.INTOIP &= ~(1 << 1);
TN_BFA(TN_BFA_CLR, IPC0, INTOIP, (1 << 1));

/-- Invert two less-significant bits of the INTOIP field
// in the IPC0 register:
// IPC0bits.INTOIP ^= 0x03;
TN_BFA(TN_BFA_INV, IPC0, INTOIP, 0x03);

/-- Write value 0x05 to the INTOIP field of the IPC0 register:
// IPC0bits.INTOIP = 0x05;
TN_BFA(TN_BFA_WR, IPC0, INTOIP, 0x05);

/-- Write value of the variable a to the INTOIP field of the IPC0
// register:
// IPC0bits.INTOIP = a;
TN_BFA(TN_BFA_WR, IPC0, INTOIP, a);

/-- Read the value that is stored in the INTOIP field of the IPC0
// register, to the int variable a:
// int a = IPC0bits.INTOIP;
a = TN_BFA(TN_BFA_RD, IPC0, INTOIP);
```

Definition at line 154 of file tn\_arch\_pic24\_bfa.h.

18.3.2.7 #define TN\_BFAR( *comm*, *reg\_name*, *lower*, *upper*, ... )

Macro for atomic access to the structure bit field specified as a range.

## Parameters

<i>comm</i>	command to execute: <ul style="list-style-type: none"> <li>• <a href="#">TN_BFA_WR</a> - write bit field</li> <li>• <a href="#">TN_BFA_RD</a> - read bit field</li> <li>• <a href="#">TN_BFA_SET</a> - set bits by mask</li> <li>• <a href="#">TN_BFA_CLR</a> - clear bits by mask</li> <li>• <a href="#">TN_BFA_INV</a> - invert bits by mask</li> </ul>
<i>reg_name</i>	variable name (PORTA, CMCON, ...). Variable should be in the near memory (first 8 KB)
<i>lower</i>	number of lowest affected bit of the field
<i>upper</i>	number of highest affected bit of the field
...	used if only <i>comm</i> != <a href="#">TN_BFA_RD</a> . Meaning depends on the <i>comm</i> , see comments for specific command: <a href="#">TN_BFA_WR</a> , etc.

## Usage examples:

```
int a = 0x02;

//-- Write constant 0xaa to the least significant byte of the TRISB
//   register:
TN_BFAR(TN_BFA_WR, TRISB, 0, 7, 0xaa);

//-- Invert least significant nibble of the most significant byte
//   in the register TRISB:
TN_BFAR(TN_BFA_INV, TRISB, 8, 15, 0x0f);

//-- Get 5 least significant bits from the register TRISB and store
//   result to the variable a
a = TN_BFAR(TN_BFA_RD, TRISB, 0, 4);
```

Definition at line 270 of file `tn_arch_pic24_bfa.h`.

## 18.4 arch/pic32/tn\_arch\_pic32.h File Reference

### 18.4.1 Detailed Description

PIC32 architecture-dependent routines.

Definition in file [tn\\_arch\\_pic32.h](#).

### Macros

- `#define tn\_p32\_soft\_isr(vec)`  
*Interrupt handler wrapper macro for software context saving.*
- `#define tn\_p32\_srs\_isr(vec)`  
*Interrupt handler wrapper macro for shadow register context saving.*
- `#define tn\_soft\_isr tn\_p32\_soft\_isr`  
*For compatibility with old projects, old name of [tn\\_p32\\_soft\\_isr\(\)](#) macro is kept; please don't use it in new code.*
- `#define tn\_srs\_isr tn\_p32\_srs\_isr`  
*For compatibility with old projects, old name of [tn\\_p32\\_srs\\_isr\(\)](#) macro is kept; please don't use it in new code.*

### Variables

- volatile int [tn\\_p32\\_int\\_nest\\_count](#)

- current interrupt nesting count.*
- void \* [tn\\_p32\\_user\\_sp](#)
  - saved task stack pointer.*
- void \* [tn\\_p32\\_int\\_sp](#)
  - saved ISR stack pointer.*

## 18.4.2 Macro Definition Documentation

### 18.4.2.1 #define tn\_p32\_soft\_isr( vec )

Interrupt handler wrapper macro for software context saving.

Usage looks like the following:

```
tn_p32_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

<i>vec</i>	interrupt vector number, such as <code>_TIMER_1_VECTOR</code> , etc.
------------	--

Definition at line 326 of file `tn_arch_pic32.h`.

### 18.4.2.2 #define tn\_p32\_srs\_isr( vec )

Interrupt handler wrapper macro for shadow register context saving.

Usage looks like the following:

```
tn_p32_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

<i>vec</i>	interrupt vector number, such as <code>_TIMER_1_VECTOR</code> , etc.
------------	--

Definition at line 475 of file `tn_arch_pic32.h`.

## 18.4.3 Variable Documentation

### 18.4.3.1 volatile int tn\_p32\_int\_nest\_count

current interrupt nesting count.

Used by macros [tn\\_p32\\_soft\\_isr\(\)](#) and [tn\\_p32\\_srs\\_isr\(\)](#).

### 18.4.3.2 void\* tn\_p32\_user\_sp

saved task stack pointer.

Needed when switching stack pointer from task stack to interrupt stack.

### 18.4.3.3 void\* tn\_p32\_int\_sp

saved ISR stack pointer.

Needed when switching stack pointer from interrupt stack to task stack.

## 18.5 arch/tn\_arch.h File Reference

### 18.5.1 Detailed Description

Architecture-dependent routines declaration.

Definition in file [tn\\_arch.h](#).

### Functions

- void [tn\\_arch\\_int\\_dis](#) (void)  
*Unconditionally disable system interrupts.*
- void [tn\\_arch\\_int\\_en](#) (void)  
*Unconditionally enable interrupts.*
- [TN\\_UWord tn\\_arch\\_sr\\_save\\_int\\_dis](#) (void)  
*Disable system interrupts and return previous value of status register, atomically.*
- void [tn\\_arch\\_sr\\_restore](#) ([TN\\_UWord](#) sr)  
*Restore previously saved status register.*
- void [\\_tn\\_arch\\_sys\\_init](#) ([TN\\_UWord](#) \*int\_stack, unsigned int int\_stack\_size)  
*Architecture-dependent system startup routine.*
- [TN\\_UWord \\*](#) [\\_tn\\_arch\\_stack\\_top\\_get](#) ([TN\\_UWord](#) \*stack\_low\_address, int stack\_size)  
*Should return top of the stack, which may be:*
- [TN\\_UWord \\*](#) [\\_tn\\_arch\\_stack\\_init](#) ([TN\\_TaskBody](#) \*task\_func, [TN\\_UWord](#) \*stack\_top, int stack\_size, void \*param)  
*Should put initial CPU context to the provided stack pointer for new task and return current stack pointer.*
- int [\\_tn\\_arch\\_inside\\_isr](#) (void)  
*Should return 1 if system ISR is currently running, 0 otherwise.*
- int [\\_tn\\_arch\\_is\\_int\\_disabled](#) (void)  
*Should return 1 if system interrupts are currently disabled, 0 otherwise.*
- void [\\_tn\\_arch\\_context\\_switch\\_pend](#) (void)  
*Called whenever we need to switch context from one task to another.*
- void [\\_tn\\_arch\\_context\\_switch\\_now\\_nosave](#) (void)  
*Called whenever we need to switch context to new task, but don't save current context.*

### 18.5.2 Function Documentation

#### 18.5.2.1 void tn\_arch\_int\_dis ( void )

Unconditionally disable *system interrupts*.

Refer to the section [Interrupt types](#) for details on what is *system interrupt*.

#### 18.5.2.2 void tn\_arch\_int\_en ( void )

Unconditionally enable interrupts.

Refer to the section [Interrupt types](#) for details on what is *system interrupt*.

## 18.5.2.3 TN\_UWord tn\_arch\_sr\_save\_int\_dis ( void )

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

Refer to the section [Interrupt types](#) for details on what is *system interrupt*.

See also

[tn\\_arch\\_sr\\_restore\(\)](#)

## 18.5.2.4 void tn\_arch\_sr\_restore ( TN\_UWord sr )

Restore previously saved status register.

Parameters

<i>sr</i>	status register value previously from <a href="#">tn_arch_sr_save_int_dis()</a>
-----------	---

See also

[tn\\_arch\\_sr\\_save\\_int\\_dis\(\)](#)

## 18.5.2.5 void \_tn\_arch\_sys\_init ( TN\_UWord \* int\_stack, unsigned int int\_stack\_size )

Architecture-dependent system startup routine.

Called from [tn\\_sys\\_start\(\)](#).

## 18.5.2.6 TN\_UWord\* \_tn\_arch\_stack\_top\_get ( TN\_UWord \* stack\_low\_address, int stack\_size )

Should return top of the stack, which may be:

- (stack\_low\_address - 1)
- (stack\_low\_address + stack\_size)
- (stack\_low\_address)
- (stack\_low\_address + stack\_size - 1)

(depending on the architecture)

Parameters

<i>stack_low</i> ↔ <i>address</i>	Start address of the stack array.
<i>stack_size</i>	Size of the stack in <a href="#">TN_UWord</a> -s, not in bytes.

## 18.5.2.7 TN\_UWord\* \_tn\_arch\_stack\_init ( TN\_TaskBody \* task\_func, TN\_UWord \* stack\_top, int stack\_size, void \* param )

Should put initial CPU context to the provided stack pointer for new task and return current stack pointer.

When resulting context gets restored by [\\_tn\\_arch\\_context\\_switch\\_now\\_nosave\(\)](#) or [\\_tn\\_arch\\_context\\_switch\\_pend\(\)](#), the following conditions should be met:

- Interrupts are enabled;

- Return address is set to `tn_task_exit()`, so that when task body function returns, `tn_task_exit()` gets automatically called;
- Argument 0 contains `param` pointer

#### Parameters

<code>task_func</code>	Pointer to task body function.
<code>stack_top</code>	Top of the stack, returned by <code>_tn_arch_stack_top_get()</code> .
<code>stack_size</code>	Size of the stack in <code>TN_UWord</code> -s, not in bytes.
<code>param</code>	User-provided parameter for task body function.

#### Returns

current stack pointer (top of the stack)

#### 18.5.2.8 `int_tn_arch_inside_isr( void )`

Should return 1 if *system ISR* is currently running, 0 otherwise.

Refer to the section [Interrupt types](#) for details on what is *system ISR*.

#### 18.5.2.9 `int_tn_arch_is_int_disabled( void )`

Should return 1 if *system interrupts* are currently disabled, 0 otherwise.

Refer to the section [Interrupt types](#) for details on what is *system interrupt*.

#### 18.5.2.10 `void_tn_arch_context_switch_pend( void )`

Called whenever we need to switch context from one task to another.

This function typically does NOT switch context; it merely pends it, that is, it sets appropriate interrupt flag. If current level is an application level, interrupt is fired immediately, and context gets switched.

But, if it's hard or impossible on particular platform to use dedicated interrupt flag, this function may just switch the context on its own.

#### 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 in actual context switching routine:

- save context of the preempted task to its stack;
- set `tn_curr_run_task` to `tn_next_task_to_run`;
- restore context of the newly activated task from its stack.

#### See also

`tn_curr_run_task`  
`tn_next_task_to_run`



## 18.5.2.11 void tn\_arch\_context\_switch\_now\_nosave ( void )

Called whenever we need to switch context to new task, but don't save current context.

This happens:

- At system start, inside `tn_sys_start()`;
- At task exit, inside `tn_task_exit()`

This function doesn't pend context switch, it switches context immediately.

**Preconditions:**

- interrupts are disabled;
- `tn_next_task_to_run` is already set to needed task.

**Actions to perform:**

- set `tn_curr_run_task` to `tn_next_task_to_run`;
- restore context of the newly activated task from its stack.

**See also**

```
tn_curr_run_task
tn_next_task_to_run
```

## 18.6 core/tn\_cfg\_dispatch.h File Reference

### 18.6.1 Detailed Description

Dispatch configuration: set predefined options, include user-provided cfg file as well as default cfg file.

Definition in file [tn\\_cfg\\_dispatch.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))`.*

### 18.6.2 Macro Definition Documentation

18.6.2.1 `#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_cfg_dispatch.h`.

### 18.6.2.2 #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_cfg_dispatch.h`.

## 18.7 core/tn\_common.h File Reference

### 18.7.1 Detailed Description

Definitions used through the whole kernel.

Definition in file `tn_common.h`.

### Macros

- #define `TN_NULL` ((void \*)0)  
*NULL pointer definition.*
- #define `TN_BOOL` int  
*boolean type definition*
- #define `TN_TRUE` (1 == 1)  
*true value definition for type TN\_BOOL*
- #define `TN_FALSE` (1 == 0)  
*false value definition for type TN\_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` = (unsigned int)0x47ABCF69, `TN_ID_SEMAPHORE` = (unsigned int)0x6FA173EB, `TN_ID_←`  
`EVENTGRP` = (unsigned int)0x5E224F25, `TN_ID_DATAQUEUE` = (unsigned int)0x8C8A6C89,  
`TN_ID_FSMEMORYPOOL` = (unsigned int)0x26B7CE8B, `TN_ID_MUTEX` = (unsigned int)0x17129E45, `T_←`  
`N_ID_TIMER` = (unsigned int)0x9A937FBC, `TN_ID_EXCHANGE` = (unsigned int)0x32b7c072,  
`TN_ID_EXCHANGE_LINK` = (unsigned int)0x24d36f35 }  
*Magic number for object validity verification TODO: use TN\_UWord here instead of unsigned int.*
- 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.*

## 18.7.2 Macro Definition Documentation

### 18.7.2.1 #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_fmем_create()` for usage example.

Definition at line 219 of file `tn_common.h`.

### 18.7.2.2 #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 243 of file `tn_common.h`.

## 18.7.3 Typedef Documentation

### 18.7.3.1 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()`. In this case, `TN_RC_FORCED` is returned by the waiting function. (the usage of `tn_task_release_wait()` function is discouraged though)

Definition at line 183 of file `tn_common.h`.

## 18.7.4 Enumeration Type Documentation

### 18.7.4.1 enum TN\_ObjId

Magic number for object validity verification TODO: use TN\_UWord here instead of unsigned int.

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  
**TN\_ID\_TIMER** id for timers  
**TN\_ID\_EXCHANGE** id for exchange objects  
**TN\_ID\_EXCHANGE\_LINK** id for exchange objects

Definition at line 65 of file tn\_common.h.

### 18.7.4.2 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\\_fmем.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 80 of file [tn\\_common.h](#).

## 18.8 core/tn\_dqueue.h File Reference

### 18.8.1 Detailed Description

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

For the useful pattern on how to use queue together with [fixed memory pool](#), refer to the example: `examples/queue`. Be sure to examine the readme there.

TNeoKernel offers a way to wait for a message from multiple queues in just a single call, refer to the section [Connecting an event group to other system objects](#) for details. Related queue services:

- [tn\\_queue\\_eventgrp\\_connect\(\)](#)
- [tn\\_queue\\_eventgrp\\_disconnect\(\)](#)

There is an example project available that demonstrates event group connection technique: `examples/queue_eventgrp_conn`. Be sure to examine the readme there.

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.*
- enum [TN\\_RCode](#) [tn\\_queue\\_eventgrp\\_connect](#) (struct [TN\\_DQueue](#) \*dque, struct [TN\\_EventGrp](#) \*eventgrp, [TN\\_UWord](#) pattern)  
*Connect an event group to the queue.*
- enum [TN\\_RCode](#) [tn\\_queue\\_eventgrp\\_disconnect](#) (struct [TN\\_DQueue](#) \*dque)  
*Disconnect a connected event group from the queue.*

## 18.8.2 Function Documentation

### 18.8.2.1 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

<i>dque</i>	pointer to already allocated struct <a href="#">TN_DQueue</a> .
<i>data_fifo</i>	pointer to already allocated array of <code>void *</code> to store data queue items. Can be <a href="#">TN_NULL</a> .
<i>items_cnt</i>	capacity of queue (count of elements in the <code>data_fifo</code> 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](#).

18.8.2.2 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.



(refer to [Legend](#) for details)

## Parameters

<i>dque</i>	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](#).

18.8.2.3 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

<i>dque</i>	pointer to data queue to send data to
<i>p_data</i>	value to send
<i>timeout</i>	refer to <a href="#">TN_Timeout</a>

## 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](#)

18.8.2.4 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)

18.8.2.5 `enum TN_RCode tn_queue_isend_polling ( struct TN_DQueue * dqe, void * p_data )`

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



(refer to [Legend](#) for details)

18.8.2.6 `enum TN_RCode tn_queue_receive ( struct TN_DQueue * dqe, void ** pp_data, TN_Timeout timeout )`

Receive the data element from the data queue specified by the `dqe` 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

<code>dqe</code>	pointer to data queue to receive data from
<code>pp_data</code>	pointer to location to store the value
<code>timeout</code>	refer to <a href="#">TN_Timeout</a>

#### 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](#)

18.8.2.7 `enum TN_RCode tn_queue_receive_polling ( struct TN_DQueue * dqe, void ** pp_data )`

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



(refer to [Legend](#) for details)

18.8.2.8 `enum TN_RCode tn_queue_ireceive_polling ( struct TN_DQueue * dqe, void ** pp_data )`

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



(refer to [Legend](#) for details)

18.8.2.9 `enum TN_RCode tn_queue_eventgrp_connect ( struct TN_DQueue * dqe, struct TN_EventGrp * eventgrp, TN_UWord pattern )`

Connect an event group to the queue.



Refer to the section [Connecting an event group to other system objects](#) for details.

Only one event group can be connected to the queue at a time. If you connect event group while another event group is already connected, the old link is discarded.

#### Parameters

<i>dque</i>	queue to which event group should be connected
<i>eventgrp</i>	event group to connect
<i>pattern</i>	flags pattern that should be managed by the queue automatically



(refer to [Legend](#) for details)

#### 18.8.2.10 enum TN\_RCode tn\_queue\_eventgrp\_disconnect ( struct TN\_DQueue \* dque )

Disconnect a connected event group from the queue.

Refer to the section [Connecting an event group to other system objects](#) for details.

If there is no event group connected, nothing is changed.

#### Parameters

<i>dque</i>	queue from which event group should be disconnected
-------------	---



(refer to [Legend](#) for details)

## 18.9 core/tn\_eventgrp.h File Reference

### 18.9.1 Detailed Description

Event group.

An event group has an internal variable (of type `TN_UWORD`), 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.

### 18.9.2 Connecting an event group to other system objects

Sometimes task needs to wait for different system events, the most common examples are:

- wait for a message from the queue(s) plus wait for some application-dependent event (such as a flag to finish the task, or whatever);
- wait for messages from multiple queues.

If the kernel doesn't offer a mechanism for that, programmer usually have to use polling services on these queues and sleep for a few system ticks. Obviously, this approach has serious drawbacks: we have a lot of useless context switches, and response for the message gets much slower. Actually, we lost the main goal of the preemptive kernel when we use polling services like that.

TNeoKernel offers a solution: an event group can be connected to other kernel objects, and these objects will maintain certain flags inside that event group automatically.

So, in case of multiple queues, we can act as follows (assume we have two queues: Q1 and Q2) :

- create event group EG;
- connect EG with flag 1 to Q1;
- connect EG with flag 2 to Q2;
- when task needs to receive a message from either Q1 or Q2, it just waits for the any of flags 1 or 2 in the EG, this is done in the single call to `tn_eventgrp_wait()`.
- when that event happened, task checks which flag is set, and receive message from the appropriate queue.

Please note that task waiting for the event should **not** clear the flag manually: this flag is maintained completely by the queue. If the queue is non-empty, the flag is set. If the queue becomes empty, the flag is cleared.

For the information on system services related to queue, refer to the [queue reference](#).

There is an example project available that demonstrates event group connection technique: `examples/queue←_eventgrp_conn`. Be sure to examine the readme there.

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`.*
- struct `TN_EGrpLink`  
*A link to event group: used when event group can be connected to some kernel object, such as queue.*

## 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_TOGGLE` }  
*Modify operation: set, clear or toggle.*

## Functions

- enum `TN_RCode` `tn_eventgrp_create` (struct `TN_EventGrp` \*eventgrp, `TN_UWord` 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, `TN_UWord` wait\_pattern, enum `TN_E←GrpWaitMode` wait\_mode, `TN_UWord` \*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, `TN_UWord` wait\_pattern, enum `TN_E←GrpWaitMode` wait\_mode, `TN_UWord` \*p\_flags\_pattern)  
*The same as `tn_eventgrp_wait()` with zero timeout.*
- enum `TN_RCode` `tn_eventgrp_await_polling` (struct `TN_EventGrp` \*eventgrp, `TN_UWord` wait\_pattern, enum `TN_E←GrpWaitMode` wait\_mode, `TN_UWord` \*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, `TN_←UWord` pattern)

Modify current events bit pattern in the event group.

- enum `TN_RCode` `tn_eventgrp_imodify` (struct `TN_EventGrp` \*eventgrp, enum `TN_EGrpOp` operation, `TN_UWord` pattern)

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

### 18.9.3 Enumeration Type Documentation

#### 18.9.3.1 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 124 of file `tn_eventgrp.h`.

#### 18.9.3.2 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 139 of file `tn_eventgrp.h`.

### 18.9.4 Function Documentation

#### 18.9.4.1 enum `TN_RCode` `tn_eventgrp_create` ( struct `TN_EventGrp` \* eventgrp, `TN_UWord` 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

<i>eventgrp</i>	Pointer to already allocated struct <code>TN_EventGrp</code>
<i>initial_pattern</i>	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`.

#### 18.9.4.2 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

<i>eventgrp</i>	Pointer to event group 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](#).

#### 18.9.4.3 enum TN\_RCode tn\_eventgrp\_wait ( struct TN\_EventGrp \* eventgrp, TN\_UWord wait\_pattern, enum TN\_EGrpWaitMode wait\_mode, TN\_UWord \* 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

<i>eventgrp</i>	Pointer to event group to wait events from
<i>wait_pattern</i>	Events bit pattern for which task should wait
<i>wait_mode</i>	Specifies whether task should wait for <b>all</b> the event bits from <code>wait_pattern</code> to be set, or for just <b>any</b> of them (see enum <a href="#">TN_EGrpWaitMode</a> )
<i>p_flags_pattern</i>	Pointer to the <code>TN_UWord</code> variable in which actual event pattern that caused task to stop waiting will be stored. May be <code>TN_NULL</code> .
<i>timeout</i>	refer to <a href="#">TN_Timeout</a>

##### 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 `TN_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](#).

#### 18.9.4.4 enum TN\_RCode tn\_eventgrp\_wait\_polling ( struct TN\_EventGrp \* eventgrp, TN\_UWord wait\_pattern, enum TN\_EGrpWaitMode wait\_mode, TN\_UWord \* p\_flags\_pattern )

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



(refer to [Legend](#) for details)

18.9.4.5 `enum TN_RCode tn_eventgrp_await_polling ( struct TN_EventGrp * eventgrp, TN_UWord wait_pattern, enum TN_EGrpWaitMode wait_mode, TN_UWord * p_flags_pattern )`

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



(refer to [Legend](#) for details)

18.9.4.6 `enum TN_RCode tn_eventgrp_modify ( struct TN_EventGrp * eventgrp, enum TN_EGrpOp operation, TN_UWord 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

<i>eventgrp</i>	Pointer to event group to modify events in
<i>operation</i>	Actual operation to perform: set, clear or toggle. Refer to enum <a href="#">TN_EGrpOp</a>
<i>pattern</i>	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](#).

18.9.4.7 `enum TN_RCode tn_eventgrp_imodify ( struct TN_EventGrp * eventgrp, enum TN_EGrpOp operation, TN_UWord pattern )`

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



(refer to [Legend](#) for details)

## 18.10 core/tn\_fmем.h File Reference

### 18.10.1 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).

For the useful pattern on how to use fixed memory pool together with [queue](#), refer to the example: `examples/queue`. Be sure to examine the readme there.

Definition in file [tn\\_fmем.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\\_fmем\\_create](#) (struct [TN\\_FMem](#) \*fmem, void \*start\_addr, unsigned int block\_size, int blocks\_cnt)

*Construct fixed memory blocks pool.*

- enum [TN\\_RCode tn\\_fmем\\_delete](#) (struct [TN\\_FMem](#) \*fmem)

*Destruct fixed memory blocks pool.*

- enum [TN\\_RCode tn\\_fmем\\_get](#) (struct [TN\\_FMem](#) \*fmem, void \*\*p\_data, [TN\\_Timeout](#) timeout)

*Get memory block from the pool.*

- enum [TN\\_RCode tn\\_fmем\\_get\\_polling](#) (struct [TN\\_FMem](#) \*fmem, void \*\*p\_data)

*The same as [tn\\_fmем\\_get\(\)](#) with zero timeout.*

- enum [TN\\_RCode tn\\_fmем\\_iget\\_polling](#) (struct [TN\\_FMem](#) \*fmem, void \*\*p\_data)

*The same as [tn\\_fmем\\_get\(\)](#) with zero timeout, but for using in the ISR.*

- enum [TN\\_RCode tn\\_fmем\\_release](#) (struct [TN\\_FMem](#) \*fmem, void \*p\_data)

*Release memory block back to the pool.*

- enum [TN\\_RCode tn\\_fmем\\_irelease](#) (struct [TN\\_FMem](#) \*fmem, void \*p\_data)

*The same as [tn\\_fmем\\_get\(\)](#), but for using in the ISR.*

## 18.10.2 Macro Definition Documentation

### 18.10.2.1 #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\\_fmем\\_create\(\)](#) for usage example.

#### Parameters

<i>name</i>	C variable name of the buffer array (this name should be given to the <a href="#">tn_fmем_create()</a> function as the <code>start_addr</code> argument)
<i>item_type</i>	Type of item in the memory pool, like <code>struct MyMemoryItem</code> .
<i>size</i>	Number of items in the memory pool.

Definition at line 140 of file `tn_fmем.h`.

### 18.10.3 Function Documentation

#### 18.10.3.1 enum TN\_RCode tn\_fmем\_create ( struct TN\_FMem \* fmem, void \* start\_addr, unsigned int block\_size, int blocks\_cnt )

Construct fixed memory blocks pool.

id\_fmем 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_fmем_buf, struct MyMemoryItem, MY_MEMORY_BUF_SIZE);

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

And then, construct your my\_fmем as follows:

```
enum TN_RCode rc;
rc = tn_fmем_create( &my_fmем,
                    my_fmем_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

<i>fmem</i>	pointer to already allocated struct <a href="#">TN_FMem</a> .
<i>start_addr</i>	pointer to start of the array; should be aligned properly, see example above
<i>block_size</i>	size of memory block; should be a multiple of sizeof(TN_UWord), see example above
<i>blocks_cnt</i>	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](#)

#### 18.10.3.2 enum TN\_RCode tn\_fmем\_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

<i>fmem</i>	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](#).

18.10.3.3 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

<i>fmem</i>	Pointer to memory pool
<i>p_data</i>	Address of the (void *) to which received block address will be saved
<i>timeout</i>	Refer to <a href="#">TN_Timeout</a>

## 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](#).

18.10.3.4 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)

18.10.3.5 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)

18.10.3.6 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

<i>fmem</i>	Pointer to memory pool.
<i>p_data</i>	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](#).

18.10.3.7 `enum TN_RCode tn_fmem_release ( struct TN_FMem * fmem, void * p_data )`

The same as [tn\\_fmem\\_get \( \)](#), but for using in the ISR.



(refer to [Legend](#) for details)

## 18.11 core/tn\_list.h File Reference

### 18.11.1 Detailed Description

Circular doubly linked list, for internal kernel usage.

Definition in file [tn\\_list.h](#).

#### Data Structures

- struct [TN\\_ListItem](#)  
*Circular doubly linked list item, for internal kernel usage.*

## 18.12 core/tn\_mutex.h File Reference

### 18.12.1 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 prevent deadlocks, although the kernel can notify you if a deadlock has occurred (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.*

## 18.12.2 Enumeration Type Documentation

### 18.12.2.1 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](#).

### 18.12.3 Function Documentation

18.12.3.1 `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

<i>mutex</i>	Pointer to already allocated <code>struct TN_Mutex</code>
<i>protocol</i>	Mutex protocol: priority ceiling or priority inheritance. See <code>enum TN_MutexProtocol</code> .
<i>ceil_priority</i>	Used if only <code>protocol</code> is <code>TN_MUTEX_PROT_CEILING</code> : 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`.

18.12.3.2 `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

<i>mutex</i>	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`.

18.12.3.3 `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

<i>mutex</i>	mutex to lock
<i>timeout</i>	refer to <a href="#">TN_Timeout</a>

## 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](#)

18.12.3.4 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)

18.12.3.5 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 or 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](#).

## 18.13 core/tn\_oldsymbols.h File Reference

### 18.13.1 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 automatically if the option `TN_OLD_TNKERNEL_NAMES` is set.

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_sem_acquire` `tn_sem_wait`
  - old name of `tn_sem_wait`*
- #define `tn_sem_acquire_polling` `tn_sem_wait_polling`
  - old name of `tn_sem_wait_polling`*
- #define `tn_sem_iacquire_polling` `tn_sem_iwait_polling`
  - old name of `tn_sem_iwait_polling`*
- #define `tn_fmем_get_ipolling` `tn_fmем_iget_polling`
  - old TNKernel name of `tn_fmем_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)`*
- #define `TN_TASK_STACK_DEF TN_STACK_ARR_DEF`  
*old name for `TN_STACK_ARR_DEF`*

## 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`*

## 18.13.2 Macro Definition Documentation

### 18.13.2.1 #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 140 of file [tn\\_oldsymbols.h](#).

## 18.14 core/tn\_sem.h File Reference

### 18.14.1 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 waited for 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\\_wait](#) (struct [TN\\_Sem](#) \*sem, [TN\\_Timeout](#) timeout)  
*Wait for the semaphore.*
- enum [TN\\_RCode tn\\_sem\\_wait\\_polling](#) (struct [TN\\_Sem](#) \*sem)



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

- enum `TN_RCode tn_sem_await_polling` (struct `TN_Sem *sem`)

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

## 18.14.2 Function Documentation

### 18.14.2.1 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

<code>sem</code>	Pointer to already allocated struct <code>TN_Sem</code>
<code>start_count</code>	Initial counter value, typically it is equal to <code>max_count</code>
<code>max_count</code>	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`.

### 18.14.2.2 enum `TN_RCode tn_sem_delete` ( struct `TN_Sem * sem` )

Destruct the semaphore.

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



(refer to [Legend](#) for details)

#### Parameters

<code>sem</code>	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`.

### 18.14.2.3 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, and first task (if any) that [waits](#) for the semaphore becomes runnable with `TN_RC_OK` returned from `tn_sem_wait()`.

if semaphore counter is already has its max value, no action performed and `TN_RC_OVERFLOW` is returned



(refer to [Legend](#) for details)

## Parameters

<i>sem</i>	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](#).

18.14.2.4 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)

18.14.2.5 enum [TN\\_RCode](#) `tn_sem_wait ( struct TN_Sem * sem, TN_Timeout timeout )`

Wait for 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: task might switch to [WAIT](#) state until someone [signaled](#) the semaphore or until the `timeout` expired. refer to [TN\\_Timeout](#).



(refer to [Legend](#) for details)

## Parameters

<i>sem</i>	semaphore to wait for
<i>timeout</i>	refer to <a href="#">TN_Timeout</a>

## Returns

- [TN\\_RC\\_OK](#) if waiting was successful
- 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](#).

18.14.2.6 enum [TN\\_RCode](#) `tn_sem_wait_polling ( struct TN_Sem * sem )`

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



(refer to [Legend](#) for details)

18.14.2.7 enum [TN\\_RCode](#) `tn_sem_await_polling ( struct TN_Sem * sem )`

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



(refer to [Legend](#) for details)

## 18.15 core/tn\_sys.h File Reference

### 18.15.1 Detailed Description

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

Definition in file [tn\\_sys.h](#).

#### Macros

- #define [TN\\_STACK\\_ARR\\_DEF](#)(name, size)  
*Convenience macro for the definition of stack array.*
- #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](#))([TN\\_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\\_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 [TN\\_BOOL](#) [tn\\_is\\_task\\_context](#) (void)

*Returns whether current system context is [TN\\_CONTEXT\\_TASK](#)*
- static [TN\\_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.*

## 18.15.2 Macro Definition Documentation

### 18.15.2.1 #define TN\_STACK\_ARR\_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

<i>name</i>	C variable name of the array
<i>size</i>	size of the stack array in words ( <a href="#">TN_UWord</a> ), not in bytes.

Definition at line 85 of file [tn\\_sys.h](#).

## 18.15.3 Typedef Documentation

### 18.15.3.1 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 162 of file tn\_sys.h.

### 18.15.3.2 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 177 of file tn\_sys.h.

### 18.15.3.3 typedef void( TN\_CBDeadlock)(TN\_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

<i>active</i>	Boolean value indicating whether deadlock becomes active or inactive. Note: deadlock might become inactive if, for example, one of tasks involved in deadlock exits from waiting by timeout.
<i>mutex</i>	mutex that is involved in deadlock. You may find out other mutexes involved by means of <code>mutex-&gt;deadlock_list</code> .
<i>task</i>	task that is involved in deadlock. You may find out other tasks involved by means of <code>task-&gt;deadlock_list</code> .

Definition at line 197 of file tn\_sys.h.

## 18.15.4 Enumeration Type Documentation

### 18.15.4.1 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\\_DEADLOCK\\_DETECT](#) is non-zero.

See also

[TN\\_MUTEX\\_DEADLOCK\\_DETECT](#)

Definition at line 100 of file tn\_sys.h.

### 18.15.4.2 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 116 of file [tn\\_sys.h](#).

## 18.15.5 Function Documentation

18.15.5.1 `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

<code>idle_task_stack</code>	Pointer to array for idle task stack. User must either use the macro <a href="#">TN_STACK_ARR_DE↔F()</a> for the definition of stack array, or allocate it manually as an array of <a href="#">TN_UWord</a> with <a href="#">TN_ARCH_STK_ATTR_BEFORE</a> and <a href="#">TN_ARCH_STK_ATTR_AFTER</a> macros.
<code>idle_task_↔stack_size</code>	Size of idle task stack, in words ( <a href="#">TN_UWord</a> )
<code>int_stack</code>	Pointer to array for interrupt stack. User must either use the macro <a href="#">TN_STACK_ARR_DE↔F()</a> for the definition of stack array, or allocate it manually as an array of <a href="#">TN_UWord</a> with <a href="#">TN_ARCH_STK_ATTR_BEFORE</a> and <a href="#">TN_ARCH_STK_ATTR_AFTER</a> macros.
<code>int_stack_size</code>	Size of interrupt stack, in words ( <a href="#">TN_UWord</a> )
<code>cb_user_task_↔create</code>	Callback function that should create initial user's task, see <a href="#">TN_CBUserTaskCreate</a> for details.
<code>cb_idle</code>	Callback function repeatedly called from idle task, see <a href="#">TN_CBIdle</a> for details.

18.15.5.2 `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)

Among other things, expired [timers](#) are fired from this function.

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.

## 18.15.5.3 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

<i>priority</i>	Priority of tasks for which time slice value should be set
<i>ticks</i>	Time slice value, in ticks. Set to <a href="#">TN_NO_TIME_SLICE</a> for no round-robin scheduling for given priority (it's default value). Value can't be higher than <a href="#">TN_MAX_TIME_SLICE</a> .

## Returns

- [TN\\_RC\\_OK](#) on success;
- [TN\\_RC\\_WCONTEXT](#) if called from wrong context;
- [TN\\_RC\\_WPARAM](#) if given `priority` or `ticks` are invalid.

## 18.15.5.4 unsigned int tn\_sys\_time\_get ( void )

Get current system ticks count.



(refer to [Legend](#) for details)

## Returns

Current system ticks count.

## 18.15.5.5 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 from `main()`, before `tn_sys_start()`.

## Parameters

<i>cb</i>	Pointer to user-provided callback function.
-----------	---

## See also

[TN\\_MUTEX\\_DEADLOCK\\_DETECT](#)  
[TN\\_CBDeadlock](#) for callback function prototype

## 18.15.5.6 enum TN\_StateFlag tn\_sys\_state\_flags\_get ( void )

Returns current system state flags.



(refer to [Legend](#) for details)

## 18.15.5.7 enum TN\_Context tn\_sys\_context\_get ( void )

Returns system context: task or ISR.



(refer to [Legend](#) for details)

**See also**

enum [TN\\_Context](#)

**18.15.5.8 static TN\_BOOL tn\_is\_task\_context ( void ) [inline],[static]**

Returns whether current system context is [TN\\_CONTEXT\\_TASK](#)



(refer to [Legend](#) for details)

**Returns**

TN\_TRUE if current system context is [TN\\_CONTEXT\\_TASK](#), TN\_FALSE otherwise.

**See also**

[tn\\_sys\\_context\\_get \(\)](#)  
enum [TN\\_Context](#)

Definition at line 376 of file [tn\\_sys.h](#).

**18.15.5.9 static TN\_BOOL tn\_is\_isr\_context ( void ) [inline],[static]**

Returns whether current system context is [TN\\_CONTEXT\\_ISR](#)



(refer to [Legend](#) for details)

**Returns**

TN\_TRUE if current system context is [TN\\_CONTEXT\\_ISR](#), TN\_FALSE otherwise.

**See also**

[tn\\_sys\\_context\\_get \(\)](#)  
enum [TN\\_Context](#)

Definition at line 395 of file [tn\\_sys.h](#).

**18.15.5.10 struct TN\_Task\* tn\_cur\_task\_get ( void )**

Returns pointer to the currently running task.



(refer to [Legend](#) for details)

**18.15.5.11 TN\_TaskBody\* tn\_cur\_task\_body\_get ( void )**

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



(refer to [Legend](#) for details)



## 18.16 core/tn\_tasks.h File Reference

### 18.16.1 Detailed Description

#### 18.16.2 Task

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

Actually, the term *thread* is more accurate than *task*, but the term *task* historically was used in TNKernel, so TNeoKernel keeps this convention.

When kernel decides that it's time to run another task, it performs *context switch*: current context (at least, values of all registers) gets saved to the preempted task's stack, pointer to currently running task is altered as well as stack pointer, and context gets restored from the stack of newly running task.

#### 18.16.3 Task states

For list of task states and their description, refer to enum [TN\\_TaskState](#).

#### 18.16.4 Creating/starting tasks

Create task and delete task are two separate actions; although you can perform both of them in one step by passing [TN\\_TASK\\_CREATE\\_OPT\\_START](#) flag to the [tn\\_task\\_create\(\)](#) function.

#### 18.16.5 Stopping/deleting tasks

Stop task and delete task are two separate actions. If task was just stopped but not deleted, it can be just restarted again by calling [tn\\_task\\_activate\(\)](#). If task was deleted, it can't be just activated: it should be re-created by [tn\\_task\\_create\(\)](#) first.

Task stops execution when:

- it calls [tn\\_task\\_exit\(\)](#);
- it returns from its task body function (it is the equivalent to [tn\\_task\\_exit\(0\)](#))
- some other task calls [tn\\_task\\_terminate\(\)](#) passing appropriate pointer to struct [TN\\_Task](#).

#### 18.16.6 Scheduling rules

TNeoKernel always runs the most privileged task in state [RUNNABLE](#). In no circumstances can task run while there is at least one task is in the [RUNNABLE](#) state with higher priority. Task will run until:

- It becomes non-runnable (say, it may wait for something, etc)
- Some other task with higher priority becomes runnable.

Tasks with the same priority may be scheduled in round robin fashion by getting a predetermined time slice for each task with this priority. Time slice is set separately for each priority. By default, round robin is turned off for all priorities.

### 18.16.7 Idle task

TNeoKernel has one system task: an idle task, which has lowest priority. It is always in the state `RUNNABLE`, and it runs only when there are no other runnable tasks.

User can provide a callback function to be called from idle task, see `TN_CBIdle`. It is useful to bring the processor to some kind of real idle state, so that device draws less current.

Definition in file `tn_tasks.h`.

#### Data Structures

- struct `TN_Task`

*Task.*

#### 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_WAITSP = (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_REASON_EVENT`,  
`TN_WAIT_REASON_DQUE_WSEND`, `TN_WAIT_REASON_DQUE_WRECEIVE`, `TN_WAIT_REASON_MUTEX_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_TASK_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\\_state\\_get](#) (struct [TN\\_Task](#) \*task, enum [TN\\_TaskState](#) \*p\_state)  
*Get current state of the task; note that returned state is a bitmask, that is, states could be combined with each other.*
- enum [TN\\_RCode tn\\_task\\_change\\_priority](#) (struct [TN\\_Task](#) \*task, int new\_priority)  
*Set new priority for task.*

## 18.16.8 Enumeration Type Documentation

### 18.16.8.1 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\\_WAITSP](#)** 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 141 of file `tn_tasks.h`.

### 18.16.8.2 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 170 of file `tn_tasks.h`.

### 18.16.8.3 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 213 of file `tn_tasks.h`.

### 18.16.8.4 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 228 of file `tn_tasks.h`.

## 18.16.9 Function Documentation

**18.16.9.1** `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_STACK_ARR_DEF()
// for that
TN_STACK_ARR_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,
    TN_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

<code>task</code>	Ready-allocated struct <code>TN_Task</code> structure. <code>id_task</code> member should not contain <code>TN_ID_TASK</code> , otherwise <code>TN_RC_WPARAM</code> is returned.
<code>task_func</code>	Pointer to task body function.
<code>priority</code>	Priority for new task. <b>NOTE:</b> the lower value, the higher priority. Must be $> 0$ and $< (TN\_PRIORITIES\_CNT - 1)$ .
<code>task_stack_low_addr</code>	Pointer to the stack for task. User must either use the macro <code>TN_STACK_ARR_DEF()</code> for the definition of stack array, or allocate it manually as an array of <code>TN_UWord</code> with <code>TN_ARCH_STK_ATTR_BEFORE</code> and <code>TN_ARCH_STK_ATTR_AFTER</code> macros.
<code>task_stack_size</code>	Size of task stack array, in words ( <code>TN_UWord</code> ), not in bytes.
<code>param</code>	Parameter that is passed to <code>task_func</code> .
<code>opts</code>	Options for task creation, refer to enum <code>TN_TaskCreateOpt</code>

### 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](#)

18.16.9.2 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

<code>task</code>	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](#)

18.16.9.3 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

<code>task</code>	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](#)

## 18.16.9.4 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

<code>timeout</code>	Refer to <code>TN_Timeout</code>
----------------------	----------------------------------

## 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](#)

## 18.16.9.5 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

<code>task</code>	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`.

## 18.16.9.6 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)

### 18.16.9.7 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

<i>task</i>	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](#)

### 18.16.9.8 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)

### 18.16.9.9 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

<i>task</i>	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](#)

18.16.9.10 `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)

18.16.9.11 `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).

Please note that returning from task body function has the same effect as calling `tn_task_exit(0)`.



(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()`

18.16.9.12 `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

<code>task</code>	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](#).

18.16.9.13 `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

<i>task</i>	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](#).

18.16.9.14 `enum TN_RCode tn_task_state_get ( struct TN_Task * task, enum TN_TaskState * p_state )`

Get current state of the task; note that returned state is a bitmask, that is, states could be combined with each other.

Currently, only [WAIT](#) and [SUSPEND](#) states are allowed to be set together. Nevertheless, it would be probably good idea to test individual bits in the returned value instead of plain comparing values.

Note that if something goes wrong, variable pointed to by `p_state` isn't touched.



(refer to [Legend](#) for details)

## Parameters

<i>task</i>	task to get state of
<i>p_state</i>	pointer to the location where to store state of the task

## Returns

state of the task

18.16.9.15 `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.



(refer to [Legend](#) for details)

**Attention**

this function is obsolete and will probably be removed

## 18.17 core/tn\_timer.h File Reference

### 18.17.1 Detailed Description

Timer is a kernel object that is used to ask the kernel to call some user-provided function at a particular time in the future, based on the *system timer* tick.

If you need to repeatedly wake up particular task, you can create semaphore which you should *wait for* in the task, and *signal* in the timer callback (remember that you should use `tn_sem_isignal()` in this callback, since it is called from an ISR).

If you need to perform rather fast action (such as toggle some pin, or the like), consider doing that right in the timer callback, in order to avoid context switch overhead.

The timer callback approach provides ultimate flexibility.

In the spirit of TNeoKernel, timers are as lightweight as possible. That's why there is only one type of timer: the single-shot timer. If you need your timer to fire repeatedly, you can easily restart it from the timer function by the `tn_timer_start()`, so it's not a problem.

When timer fires, the user-provided function is called. Be aware of the following:

- Function is called from an ISR context (namely, from *system timer* ISR, by the `tn_tick_int_processing()`);
- Function is called with global interrupts disabled.

Consequently:

- It's legal to call interrupt services from this function;
- You should make sure that your interrupt stack is enough for this function;
- The function should be as fast as possible;
- The function should not enable interrupts unconditionally. Consider using `tn_arch_sr_save_int_dis()` and `tn_arch_sr_restore()` if you need.

See `TN_TimerFunc` for the prototype of the function that could be scheduled.

### 18.17.2 Implementation of timers

Although you don't have to understand the implementation of timers to use them, it is probably worth knowing, particularly because the kernel have an option `TN_TICK_LISTS_CNT` to customize the balance between performance of `tn_tick_int_processing()` and memory occupied by timers.

The easiest implementation of timers could be something like this: we have just a single list with all active timers, and at every system tick we should walk through all the timers in this list, and do the following with each timer:

- Decrement timeout by 1
- If new timeout is 0, then remove that timer from the list (i.e. make timer inactive), and fire the appropriate timer function.

This approach has drawbacks:

- We can't manage timers from the function called by timer. If we do so (say, if we start new timer), then the timer list gets modified. But we are currently iterating through this list, so, it's quite easy to mix things up.
- It is inefficient on rather large amount of timers and/or with large timeouts: we should iterate through all of them each system tick.

The latter is probably not so critical in the embedded world since large amount of timers is unlikely there; whereas the former is actually notable.

So, different approach was applied. The main idea is taken from the mainline Linux kernel source, but the implementation was simplified much because (1) embedded systems have much less resources, and (2) the kernel doesn't need to scale as well as Linux does. You can read about Linux timers implementation in the book "Linux Device Drivers", 3rd edition:

- Time, Delays, and Deferred Work
  - Kernel Timers
    - \* The Implementation of Kernel Timers

This book is freely available at <http://lwn.net/Kernel/LDD3/>.

So, TNeoKernel's implementation:

We have configurable value  $N$  that is a power of two, typical values are 4, 8 or 16.

If the timer expires in the next 1 to  $(N - 1)$  system ticks, it is added to one of the  $N$  lists (the so-called "tick" lists) devoted to short-range timers using the least significant bits of the `timeout` value. If it expires farther in the future, it is added to the "generic" list.

Each  $N$ -th system tick, all the timers from "generic" list are walked through, and the following is performed with each timer:

- `timeout` value decremented by  $N$
- if resulting `timeout` is less than  $N$ , timer is moved to the appropriate "tick" list.

At every system tick, all the timers from current "tick" list are fired unconditionally. This is an efficient and nice solution.

The attentive reader may want to ask why do we use  $(N - 1)$  "tick" lists if we actually have  $N$  lists. That's because, again, we want to be able to modify timers from the timer function. If we use  $N$  lists, and user wants to add new timer with `timeout` equal to  $N$ , then new timer will be added to the same list which is iterated through at the moment, and things will be mixed up.

If we use  $(N - 1)$  lists, we are guaranteed that new timers can't be added to the current "tick" list while we are iterating through it. (although timer can be deleted from that list, but it's ok)

The  $N$  in the TNeoKernel is configured by the compile-time option `TN_TICK_LISTS_CNT`.

Definition in file [tn\\_timer.h](#).

## Data Structures

- struct [TN\\_Timer](#)  
*Timer.*

## Typedefs

- typedef void( [TN\\_TimerFunc](#) )(struct [TN\\_Timer](#) \*timer, void \*p\_user\_data)  
*Prototype of the function that should be called by timer.*

## Functions

- enum [TN\\_RCode tn\\_timer\\_create](#) (struct [TN\\_Timer](#) \*timer, [TN\\_TimerFunc](#) \*func, void \*p\_user\_data)  
*Construct the timer.*
- enum [TN\\_RCode tn\\_timer\\_delete](#) (struct [TN\\_Timer](#) \*timer)  
*Destruct the timer.*
- enum [TN\\_RCode tn\\_timer\\_start](#) (struct [TN\\_Timer](#) \*timer, [TN\\_Timeout](#) timeout)  
*Start or restart the timer: that is, schedule the timer's function (given to [tn\\_timer\\_create\(\)](#)) to be called later by the kernel.*
- enum [TN\\_RCode tn\\_timer\\_cancel](#) (struct [TN\\_Timer](#) \*timer)  
*If timer is active, cancel it.*
- enum [TN\\_RCode tn\\_timer\\_set\\_func](#) (struct [TN\\_Timer](#) \*timer, [TN\\_TimerFunc](#) \*func, void \*p\_user\_data)  
*Set user-provided function and pointer to user data for the timer.*
- enum [TN\\_RCode tn\\_timer\\_is\\_active](#) (struct [TN\\_Timer](#) \*timer, [TN\\_BOOL](#) \*p\_is\_active)  
*Returns whether given timer is active or inactive.*
- enum [TN\\_RCode tn\\_timer\\_time\\_left](#) (struct [TN\\_Timer](#) \*timer, [TN\\_Timeout](#) \*p\_time\_left)  
*Returns how many [system timer](#) ticks (at most) is left for the timer to expire.*

### 18.17.3 Typedef Documentation

#### 18.17.3.1 typedef void( [TN\\_TimerFunc](#))(struct [TN\\_Timer](#) \*timer, void \*p\_user\_data)

Prototype of the function that should be called by timer.

When timer fires, the user-provided function is called. Be aware of the following:

- Function is called from ISR context (namely, from [system timer](#) ISR, by the [tn\\_tick\\_int\\_processing\(\)](#));
- Function is called with global interrupts disabled.

Consequently:

- It's legal to call interrupt services from this function;
- The function should be as fast as possible.

#### Parameters

<i>timer</i>	Timer that caused function to be called
<i>p_user_data</i>	The user-provided pointer given to <a href="#">tn_timer_create()</a> .

Definition at line 197 of file [tn\\_timer.h](#).

### 18.17.4 Function Documentation

#### 18.17.4.1 enum [TN\\_RCode tn\\_timer\\_create](#) ( struct [TN\\_Timer](#) \* timer, [TN\\_TimerFunc](#) \* func, void \* p\_user\_data )

Construct the timer.

`id_timer` field should not contain [TN\\_ID\\_TIMER](#), otherwise, [TN\\_RC\\_WPARAM](#) is returned.



(refer to [Legend](#) for details)

## Parameters

<i>timer</i>	Pointer to already allocated <code>struct TN_Timer</code>
<i>func</i>	Function to be called by timer, can't be <code>TN_NULL</code> . See <code>TN_TimerFunc()</code>
<i>p_user_data</i>	User data pointer that is given to user-provided <code>func</code> .

## Returns

- `TN_RC_OK` if timer was successfully created;
- `TN_RC_WPARAM` if wrong params were given.

18.17.4.2 `enum TN_RCode tn_timer_delete ( struct TN_Timer * timer )`

Destruct the timer.

If the timer is active, it is cancelled first.



(refer to [Legend](#) for details)

## Parameters

<i>timer</i>	timer to destruct
--------------	-------------------

## Returns

- `TN_RC_OK` if timer 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`.

18.17.4.3 `enum TN_RCode tn_timer_start ( struct TN_Timer * timer, TN_Timeout timeout )`

Start or restart the timer: that is, schedule the timer's function (given to `tn_timer_create()`) to be called later by the kernel.

See `TN_TimerFunc()`.

It is legal to restart already active timer. In this case, the timer will be cancelled first.



(refer to [Legend](#) for details)

## Parameters

<i>timer</i>	Timer to start
<i>timeout</i>	Number of system ticks after which timer should fire (i.e. function should be called). <b>Note</b> that <code>timeout</code> can't be <code>TN_WAIT_INFINITE</code> or 0.

## Returns

- `TN_RC_OK` if timer was successfully started;
- `TN_RC_WCONTEXT` if called from wrong context;
- `TN_RC_WPARAM` if wrong params were given: say, `timeout` is either `TN_WAIT_INFINITE` or 0.
- If `TN_CHECK_PARAM` is non-zero, additional return code is available: `TN_RC_INVALID_OBJ`.

18.17.4.4 enum TN\_RCode tn\_timer\_cancel ( struct TN\_Timer \* timer )

If timer is active, cancel it.

If timer is already inactive, nothing is changed.



(refer to [Legend](#) for details)

## Parameters

<i>timer</i>	Timer to cancel
--------------	-----------------

## Returns

- [TN\\_RC\\_OK](#) if timer was successfully cancelled;
- [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](#).

18.17.4.5 `enum TN_RCode tn_timer_set_func ( struct TN_Timer * timer, TN_TimerFunc * func, void * p_user_data )`

Set user-provided function and pointer to user data for the timer.

Can be called if timer is either active or inactive.



(refer to [Legend](#) for details)

## Parameters

<i>timer</i>	Pointer to timer
<i>func</i>	Function to be called by timer, can't be <a href="#">TN_NULL</a> . See <a href="#">TN_TimerFunc ()</a>
<i>p_user_data</i>	User data pointer that is given to user-provided <code>func</code> .

## Returns

- [TN\\_RC\\_OK](#) if operation was successful;
- [TN\\_RC\\_WPARAM](#) if wrong params were given.

18.17.4.6 `enum TN_RCode tn_timer_is_active ( struct TN_Timer * timer, TN_BOOL * p_is_active )`

Returns whether given timer is active or inactive.



(refer to [Legend](#) for details)

## Parameters

<i>timer</i>	Pointer to timer
<i>p_is_active</i>	Pointer to <a href="#">TN_BOOL</a> variable in which resulting value should be stored

## Returns

- [TN\\_RC\\_OK](#) if operation was successful;
- [TN\\_RC\\_WPARAM](#) if wrong params were given.

18.17.4.7 `enum TN_RCode tn_timer_time_left ( struct TN_Timer * timer, TN_Timeout * p_time_left )`

Returns how many [system timer](#) ticks (at most) is left for the timer to expire.

If timer is inactive, 0 is returned.



(refer to [Legend](#) for details)



## Parameters

<code>timer</code>	Pointer to timer
<code>p_time_left</code>	Pointer to <code>TN_Timeout</code> variable in which resulting value should be stored

## Returns

- `TN_RC_OK` if operation was successful;
- `TN_RC_WPARAM` if wrong params were given.

## 18.18 tn.h File Reference

### 18.18.1 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](#).

## 18.19 tn\_cfg\_default.h File Reference

### 18.19.1 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_PRIORITIES_CNT TN_PRIORITIES_MAX_CNT`  
*Number of priorities that can be used by application, plus one for idle task (which has the lowest priority).*
- `#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_TICK_LISTS_CNT 8`  
*Number of "tick" lists of timers, must be a power of two; minimum value: 2; typical values: 4, 8 or 16.*
- `#define TN_API_MAKE_ALIG_ARG TN_API_MAKE_ALIG_ARG__SIZE`  
*API option for `MAKE_ALIG()` macro.*
- `#define TN_P24_SYS_IPL 4`  
*Maximum system interrupt priority.*
- `#define TN_P24_SYS_IPL_STR "4"`  
*The same as `TN_P24_SYS_IPL` but should be set as string, for example:*

## 18.19.2 Macro Definition Documentation

### 18.19.2.1 `#define TN_PRIORITIES_CNT TN_PRIORITIES_MAX_CNT`

Number of priorities that can be used by application, plus one for idle task (which has the lowest priority).

This value can't be higher than architecture-dependent value `TN_PRIORITIES_MAX_CNT`, which typically equals to width of `int` type. So, for 32-bit systems, max number of priorities is 32.

But usually, application needs much less: I can imagine **at most** 4-5 different priorities, plus one for the idle task.

Do note also that each possible priority level takes RAM: two pointers for linked list and one `short` for time slice value, so on 32-bit system it takes 10 bytes. So, with default value of 32 priorities available, it takes 320 bytes. If you set it, say, to 5, you save 270 bytes, which might be notable.

Default: `TN_PRIORITIES_MAX_CNT`.

Definition at line 94 of file `tn_cfg_default.h`.

### 18.19.2.2 `#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 111 of file `tn_cfg_default.h`.

### 18.19.2.3 `#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 121 of file `tn_cfg_default.h`.

#### 18.19.2.4 #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 131 of file tn\_cfg\_default.h.

#### 18.19.2.5 #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 155 of file tn\_cfg\_default.h.

#### 18.19.2.6 #define TN\_TICK\_LISTS\_CNT 8

Number of "tick" lists of timers, must be a power of two; minimum value: 2; typical values: 4, 8 or 16.

Refer to the [Implementation of timers](#) for details.

Shortly: this value represents number of elements in the array of `struct TN_ListItem`, on 32-bit system each element takes 8 bytes.

The larger value, the more memory is needed, and the faster *system timer* ISR works. If your application has a lot of timers and/or sleeping tasks, consider incrementing this value; otherwise, default value should work for you.

Definition at line 173 of file tn\_cfg\_default.h.

#### 18.19.2.7 #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 dsPIC↔C/PIC24/PIC32 by AlexB.

Definition at line 205 of file tn\_cfg\_default.h.

#### 18.19.2.8 #define TN\_P24\_SYS\_IPL 4

Maximum system interrupt priority.

For details on system interrupts on PIC24/dsPIC, refer to the section [PIC24/dsPIC interrupts](#).

**Attention**

you should also set `TN_P24_SYS_IPL_STR` to appropriate value.

Should be  $\geq 1$  and  $\leq 6$ . Default: 4.

Definition at line 248 of file `tn_cfg_default.h`.

**18.19.2.9** `#define TN_P24_SYS_IPL_STR "4"`

The same as `TN_P24_SYS_IPL` but should be set as string, for example:

```
#define TN_P24_SYS_IPL_STR "4"
```

It is used in assembly for ISR macro `tn_p24_soft_isr()`. I don't like that we have to keep two macros instead of just one, so if anybody knows how to use integer value there, please let me know :)

Definition at line 249 of file `tn_cfg_default.h`.

# Index

TN\_CONTEXT\_ISR  
tn\_sys.h, 98

TN\_CONTEXT\_NONE  
tn\_sys.h, 98

TN\_CONTEXT\_TASK  
tn\_sys.h, 98

TN\_EVENTGRP\_OP\_CLEAR  
tn\_eventgrp.h, 79

TN\_EVENTGRP\_OP\_SET  
tn\_eventgrp.h, 79

TN\_EVENTGRP\_OP\_TOGGLE  
tn\_eventgrp.h, 79

TN\_EVENTGRP\_WMODE\_AND  
tn\_eventgrp.h, 79

TN\_EVENTGRP\_WMODE\_OR  
tn\_eventgrp.h, 79

TN\_ID\_DATAQUEUE  
tn\_common.h, 72

TN\_ID\_EVENTGRP  
tn\_common.h, 72

TN\_ID\_EXCHANGE  
tn\_common.h, 72

TN\_ID\_EXCHANGE\_LINK  
tn\_common.h, 72

TN\_ID\_FSMEMORYPOOL  
tn\_common.h, 72

TN\_ID\_MUTEX  
tn\_common.h, 72

TN\_ID\_SEMAPHORE  
tn\_common.h, 72

TN\_ID\_TASK  
tn\_common.h, 72

TN\_ID\_TIMER  
tn\_common.h, 72

TN\_MUTEX\_PROT\_CEILING  
tn\_mutex.h, 86

TN\_MUTEX\_PROT\_INHERIT  
tn\_mutex.h, 86

TN\_RC\_DELETED  
tn\_common.h, 73

TN\_RC\_FORCED  
tn\_common.h, 73

TN\_RC\_ILLEGAL\_USE  
tn\_common.h, 73

TN\_RC\_INTERNAL  
tn\_common.h, 73

TN\_RC\_INVALID\_OBJ  
tn\_common.h, 73

TN\_RC\_OK  
tn\_common.h, 72

TN\_RC\_OVERFLOW  
tn\_common.h, 72

TN\_RC\_TIMEOUT  
tn\_common.h, 72

TN\_RC\_WCONTEXT  
tn\_common.h, 72

TN\_RC\_WPARAM  
tn\_common.h, 72

TN\_RC\_WSTATE  
tn\_common.h, 72

TN\_STATE\_FLAG\_DEADLOCK  
tn\_sys.h, 97

TN\_STATE\_FLAG\_SYS\_RUNNING  
tn\_sys.h, 97

TN\_TASK\_CREATE\_OPT\_IDLE  
tn\_tasks.h, 104

TN\_TASK\_CREATE\_OPT\_START  
tn\_tasks.h, 104

TN\_TASK\_EXIT\_OPT\_DELETE  
tn\_tasks.h, 104

TN\_TASK\_STATE\_DORMANT  
tn\_tasks.h, 103

TN\_TASK\_STATE\_NONE  
tn\_tasks.h, 103

TN\_TASK\_STATE\_RUNNABLE  
tn\_tasks.h, 103

TN\_TASK\_STATE\_SUSPEND  
tn\_tasks.h, 103

TN\_TASK\_STATE\_WAIT  
tn\_tasks.h, 103

TN\_TASK\_STATE\_WAITSUSP  
tn\_tasks.h, 103

TN\_WAIT\_REASON\_DQUE\_WRECEIVE  
tn\_tasks.h, 104

TN\_WAIT\_REASON\_DQUE\_WSEND  
tn\_tasks.h, 104

TN\_WAIT\_REASON\_EVENT  
tn\_tasks.h, 103

TN\_WAIT\_REASON\_MUTEX\_C  
tn\_tasks.h, 104

TN\_WAIT\_REASON\_MUTEX\_I  
tn\_tasks.h, 104

TN\_WAIT\_REASON\_NONE  
tn\_tasks.h, 103

TN\_WAIT\_REASON\_SEM  
tn\_tasks.h, 103

TN\_WAIT\_REASON\_SLEEP  
tn\_tasks.h, 103

TN\_WAIT\_REASON\_WFIXMEM  
     tn\_tasks.h, 104  
 tn\_common.h  
     TN\_ID\_DATAQUEUE, 72  
     TN\_ID\_EVENTGRP, 72  
     TN\_ID\_EXCHANGE, 72  
     TN\_ID\_EXCHANGE\_LINK, 72  
     TN\_ID\_FSMEMORYPOOL, 72  
     TN\_ID\_MUTEX, 72  
     TN\_ID\_SEMAPHORE, 72  
     TN\_ID\_TASK, 72  
     TN\_ID\_TIMER, 72  
     TN\_RC\_DELETED, 73  
     TN\_RC\_FORCED, 73  
     TN\_RC\_ILLEGAL\_USE, 73  
     TN\_RC\_INTERNAL, 73  
     TN\_RC\_INVALID\_OBJ, 73  
     TN\_RC\_OK, 72  
     TN\_RC\_OVERFLOW, 72  
     TN\_RC\_TIMEOUT, 72  
     TN\_RC\_WCONTEXT, 72  
     TN\_RC\_WPARAM, 72  
     TN\_RC\_WSTATE, 72  
 tn\_eventgrp.h  
     TN\_EVENTGRP\_OP\_CLEAR, 79  
     TN\_EVENTGRP\_OP\_SET, 79  
     TN\_EVENTGRP\_OP\_TOGGLE, 79  
     TN\_EVENTGRP\_WMODE\_AND, 79  
     TN\_EVENTGRP\_WMODE\_OR, 79  
 tn\_mutex.h  
     TN\_MUTEX\_PROT\_CEILING, 86  
     TN\_MUTEX\_PROT\_INHERIT, 86  
 tn\_sys.h  
     TN\_CONTEXT\_ISR, 98  
     TN\_CONTEXT\_NONE, 98  
     TN\_CONTEXT\_TASK, 98  
     TN\_STATE\_FLAG\_DEADLOCK, 97  
     TN\_STATE\_FLAG\_SYS\_RUNNING, 97  
 tn\_tasks.h  
     TN\_TASK\_CREATE\_OPT\_IDLE, 104  
     TN\_TASK\_CREATE\_OPT\_START, 104  
     TN\_TASK\_EXIT\_OPT\_DELETE, 104  
     TN\_TASK\_STATE\_DORMANT, 103  
     TN\_TASK\_STATE\_NONE, 103  
     TN\_TASK\_STATE\_RUNNABLE, 103  
     TN\_TASK\_STATE\_SUSPEND, 103  
     TN\_TASK\_STATE\_WAIT, 103  
     TN\_TASK\_STATE\_WAITSUSP, 103  
     TN\_WAIT\_REASON\_DQUE\_WRECEIVE, 104  
     TN\_WAIT\_REASON\_DQUE\_WSEND, 104  
     TN\_WAIT\_REASON\_EVENT, 103  
     TN\_WAIT\_REASON\_MUTEX\_C, 104  
     TN\_WAIT\_REASON\_MUTEX\_I, 104  
     TN\_WAIT\_REASON\_NONE, 103  
     TN\_WAIT\_REASON\_SEM, 103  
     TN\_WAIT\_REASON\_SLEEP, 103  
     TN\_WAIT\_REASON\_WFIXMEM, 104