

TNeo
v1.05

Generated by Doxygen 1.8.8

Fri Jan 2 2015 03:12:24

Contents

1	TNeo overview	1
2	Foreword	3
3	Features	5
3.1	Feature list	5
4	Quick guide	7
4.1	Using TNeo in your application	7
4.2	Time ticks	7
4.3	Starting the kernel	7
4.4	Round-robin scheduling	12
5	Time ticks	13
5.1	Static tick	13
5.2	Dynamic tick	13
6	Interrupts	15
6.1	Interrupt stack	15
6.2	Interrupt types	15
7	Building TNeo	17
7.1	Configuration file	17
7.2	Makefile or library projects	17
7.2.1	Makefile	17
7.2.2	Library project	18
7.3	Building manually	18
8	Architecture-specific details	21
8.1	PIC32 port details	21
8.1.1	Context switch	21
8.1.2	Interrupts	21
8.1.3	Building	22
8.2	PIC24/dsPIC port details	22

8.2.1	Context switch	22
8.2.2	Interrupts	23
8.2.3	Atomic access to the structure bit field	24
8.2.4	Building	24
8.3	Cortex-M0/M0+/M3/M4/M4F port details	24
8.3.1	Context switch	24
8.3.2	Interrupts	24
8.3.3	Building	24
9	Why reimplement TNKernel	27
9.1	Essential problems of TNKernel	27
9.2	Examples of poor implementation	27
9.2.1	One entry point, one exit point	27
9.2.2	Don't repeat yourself	29
9.2.3	Macros that return from function	29
9.2.4	Code for doubly-linked lists	30
9.3	Bugs of TNKernel 2.7	30
10	Differences from TNKernel API	33
10.1	Incompatible API changes	33
10.1.1	System startup	33
10.1.2	Task creation API	33
10.1.3	Task wakeup count, activate count, suspend count	33
10.1.4	Fixed memory pool: non-aligned address or block size	34
10.1.5	Task service return values cleaned	34
10.1.6	Force task releasing from wait	35
10.1.7	Return code of tn_task_sleep()	35
10.1.8	Events API is changed almost completely	35
10.1.9	Zero timeout given to system functions	35
10.2	New features	35
10.3	Compatible API changes	36
10.3.1	Macro MAKE_ALIG()	36
10.3.2	Convenience macros for stack arrays definition	36
10.3.3	Convenience macros for fixed memory block pool buffers definition	36
10.3.4	Things renamed	36
10.3.5	We should wait for semaphore, not acquire it	37
10.4	Changes that do not affect API directly	37
10.4.1	No timer task	37
11	Unit tests	39
11.1	Tested CPUs	39

11.2	How tests are implemented	39
11.3	Get unit-tests	41
12	Plans	43
13	Changelog	45
13.1	v1.05	45
13.2	v1.04	46
13.3	v1.03	46
13.4	v1.02	46
13.5	v1.01	47
13.6	v1.0	47
14	Thanks	49
15	License	51
16	Legend	53
17	Data Structure Index	55
17.1	Data Structures	55
18	File Index	57
18.1	File List	57
19	Data Structure Documentation	59
19.1	_TN_BuildCfg Struct Reference	59
19.1.1	Detailed Description	59
19.2	_TN_TaskProfiler Struct Reference	60
19.2.1	Detailed Description	60
19.2.2	Field Documentation	60
19.2.2.1	last_wait_reason	60
19.2.2.2	is_running	60
19.2.2.3	timing	61
19.3	TN_DQueue Struct Reference	61
19.3.1	Detailed Description	61
19.4	TN_DQueueTaskWait Struct Reference	61
19.4.1	Detailed Description	61
19.5	TN_EGrpLink Struct Reference	62
19.5.1	Detailed Description	62
19.6	TN_EGrpTaskWait Struct Reference	62
19.6.1	Detailed Description	62
19.7	TN_EventGrp Struct Reference	63

19.7.1 Detailed Description	63
19.7.2 Field Documentation	63
19.7.2.1 attr	63
19.8 TN_FMem Struct Reference	63
19.8.1 Detailed Description	63
19.8.2 Field Documentation	64
19.8.2.1 block_size	64
19.8.2.2 start_addr	64
19.9 TN_FMemTaskWait Struct Reference	64
19.9.1 Detailed Description	64
19.10 TN_ListItem Struct Reference	64
19.10.1 Detailed Description	64
19.11 TN_Mutex Struct Reference	65
19.11.1 Detailed Description	65
19.12 TN_Sem Struct Reference	65
19.12.1 Detailed Description	65
19.13 TN_Task Struct Reference	66
19.13.1 Detailed Description	66
19.13.2 Field Documentation	67
19.13.2.1 stack_top	67
19.13.2.2 deadlock_list	67
19.13.2.3 subsys_wait	68
19.13.2.4 priority_already_updated	68
19.13.2.5 waited	68
19.14 TN_TaskTiming Struct Reference	68
19.14.1 Detailed Description	68
19.14.2 Field Documentation	69
19.14.2.1 total_run_time	69
19.14.2.2 got_running_cnt	69
19.14.2.3 total_wait_time	69
19.14.2.4 max_consecutive_wait_time	69
19.15 TN_Timer Struct Reference	69
19.15.1 Detailed Description	69
19.15.2 Field Documentation	70
19.15.2.1 start_tick_cnt	70
19.15.2.2 timeout	70
19.15.2.3 timeout_cur	70
20 File Documentation	71
20.1 arch/cortex_m/tn_arch_cortex_m.h File Reference	71

20.1.1 Detailed Description	71
20.2 arch/example/tn_arch_example.h File Reference	71
20.2.1 Detailed Description	71
20.2.2 Macro Definition Documentation	72
20.2.2.1 _TN_FFS	72
20.2.2.2 _TN_FATAL_ERROR	72
20.2.2.3 TN_ARCH_STK_ATTR_BEFORE	73
20.2.2.4 TN_ARCH_STK_ATTR_AFTER	73
20.2.2.5 TN_PRIORITIES_MAX_CNT	73
20.2.2.6 TN_INTSAVE_DATA	73
20.2.2.7 TN_INTSAVE_DATA_INT	73
20.2.2.8 TN_INT_DIS_SAVE	74
20.2.2.9 TN_INT_RESTORE	74
20.2.2.10 TN_INT_IDIS_SAVE	74
20.2.2.11 TN_INT_IRESTORE	74
20.2.2.12 _TN_SIZE_BYTES_TO_UWORDS	74
20.2.2.13 _TN_INLINE	75
20.2.2.14 _TN_VOLATILE_WORKAROUND	75
20.2.3 Typedef Documentation	75
20.2.3.1 TN_UWord	75
20.2.3.2 TN_UIntPtr	75
20.3 arch/pic24_dspic/tn_arch_pic24.h File Reference	75
20.3.1 Detailed Description	75
20.3.2 Macro Definition Documentation	75
20.3.2.1 tn_p24_soft_isr	75
20.4 arch/pic24_dspic/tn_arch_pic24_bfa.h File Reference	76
20.4.1 Detailed Description	76
20.4.2 Macro Definition Documentation	76
20.4.2.1 TN_BFA_SET	76
20.4.2.2 TN_BFA_CLR	77
20.4.2.3 TN_BFA_INV	77
20.4.2.4 TN_BFA_WR	77
20.4.2.5 TN_BFA_RD	77
20.4.2.6 TN_BFA	77
20.4.2.7 TN_BFAR	78
20.5 arch/pic32/tn_arch_pic32.h File Reference	79
20.5.1 Detailed Description	79
20.5.2 Macro Definition Documentation	80
20.5.2.1 tn_p32_soft_isr	80
20.5.2.2 tn_p32_srs_isr	80

20.5.3	Variable Documentation	80
20.5.3.1	tn_p32_int_nest_count	80
20.5.3.2	tn_p32_user_sp	80
20.5.3.3	tn_p32_int_sp	80
20.6	arch/pic32/tn_arch_pic32_bfa.h File Reference	81
20.6.1	Detailed Description	81
20.6.2	Macro Definition Documentation	81
20.6.2.1	TN_BFA_SET	81
20.6.2.2	TN_BFA_CLR	81
20.6.2.3	TN_BFA_INV	81
20.6.2.4	TN_BFA_WR	82
20.6.2.5	TN_BFA_RD	82
20.6.2.6	TN_BFA	82
20.6.2.7	TN_BFAR	83
20.7	arch/tn_arch.h File Reference	84
20.7.1	Detailed Description	84
20.7.2	Function Documentation	85
20.7.2.1	tn_arch_int_dis	85
20.7.2.2	tn_arch_int_en	85
20.7.2.3	tn_arch_sr_save_int_dis	85
20.7.2.4	tn_arch_sr_restore	85
20.7.2.5	_tn_arch_stack_top_get	86
20.7.2.6	_tn_arch_stack_bottom_empty_get	86
20.7.2.7	_tn_arch_stack_init	87
20.7.2.8	_tn_arch_inside_isr	87
20.7.2.9	_tn_arch_is_int_disabled	87
20.7.2.10	_tn_arch_context_switch_pend	87
20.7.2.11	_tn_arch_context_switch_now_nosave	88
20.7.2.12	_tn_arch_sys_start	88
20.8	core/tn_cfg_dispatch.h File Reference	88
20.8.1	Detailed Description	88
20.8.2	Macro Definition Documentation	89
20.8.2.1	TN_API_MAKE_ALIG_ARG__TYPE	89
20.8.2.2	TN_API_MAKE_ALIG_ARG__SIZE	89
20.8.2.3	_TN_ON_CONTEXT_SWITCH_HANDLER	89
20.9	core/tn_common.h File Reference	89
20.9.1	Detailed Description	89
20.9.2	Macro Definition Documentation	90
20.9.2.1	TN_MAKE_ALIG_SIZE	90
20.9.2.2	TN_MAKE_ALIG	90

20.9.3 Typedef Documentation	91
20.9.3.1 TN_TickCnt	91
20.9.4 Enumeration Type Documentation	91
20.9.4.1 TN_ObjId	91
20.9.4.2 TN_RCode	92
20.10core/tn_dqueue.h File Reference	93
20.10.1 Detailed Description	93
20.10.2 Function Documentation	94
20.10.2.1 tn_queue_create	94
20.10.2.2 tn_queue_delete	94
20.10.2.3 tn_queue_send	94
20.10.2.4 tn_queue_send_polling	96
20.10.2.5 tn_queue_isend_polling	96
20.10.2.6 tn_queue_receive	96
20.10.2.7 tn_queue_receive_polling	97
20.10.2.8 tn_queue_ireceive_polling	97
20.10.2.9 tn_queue_eventgrp_connect	97
20.10.2.10tn_queue_eventgrp_disconnect	97
20.11core/tn_eventgrp.h File Reference	98
20.11.1 Detailed Description	98
20.11.2 Connecting an event group to other system objects	98
20.11.3 Enumeration Type Documentation	99
20.11.3.1 TN_EGrpWaitMode	99
20.11.3.2 TN_EGrpOp	100
20.11.3.3 TN_EGrpAttr	100
20.11.4 Function Documentation	100
20.11.4.1 tn_eventgrp_create_wattr	100
20.11.4.2 tn_eventgrp_create	101
20.11.4.3 tn_eventgrp_delete	101
20.11.4.4 tn_eventgrp_wait	101
20.11.4.5 tn_eventgrp_wait_polling	102
20.11.4.6 tn_eventgrp_await_polling	102
20.11.4.7 tn_eventgrp_modify	102
20.11.4.8 tn_eventgrp_imodify	103
20.12core/tn_fmem.h File Reference	103
20.12.1 Detailed Description	103
20.12.2 Macro Definition Documentation	104
20.12.2.1 TN_FMEM_BUF_DEF	104
20.12.3 Function Documentation	104
20.12.3.1 tn_fmem_create	104

20.12.3.2 tn_fmem_delete	105
20.12.3.3 tn_fmem_get	105
20.12.3.4 tn_fmem_get_polling	106
20.12.3.5 tn_fmem_iget_polling	106
20.12.3.6 tn_fmem_release	106
20.12.3.7 tn_fmem_irelease	106
20.13core/tn_list.h File Reference	106
20.13.1 Detailed Description	106
20.14core/tn_mutex.h File Reference	107
20.14.1 Detailed Description	107
20.14.2 Enumeration Type Documentation	108
20.14.2.1 TN_MutexProtocol	108
20.14.3 Function Documentation	108
20.14.3.1 tn_mutex_create	108
20.14.3.2 tn_mutex_delete	108
20.14.3.3 tn_mutex_lock	109
20.14.3.4 tn_mutex_lock_polling	109
20.14.3.5 tn_mutex_unlock	110
20.15core/tn_oldsymbols.h File Reference	110
20.15.1 Detailed Description	110
20.15.2 Macro Definition Documentation	113
20.15.2.1 MAKE_ALIG	113
20.15.2.2 TN_EVENT_ATTR_SINGLE	114
20.15.2.3 TN_EVENT_ATTR_MULTI	114
20.15.2.4 TN_EVENT_ATTR_CLR	114
20.15.2.5 tn_event_create	114
20.15.2.6 tn_event_delete	114
20.15.2.7 tn_event_wait	114
20.15.2.8 tn_event_wait_polling	115
20.15.2.9 tn_event_await	115
20.15.2.10tn_event_set	115
20.15.2.11tn_event_iset	115
20.15.2.12tn_event_clear	115
20.15.2.13tn_event_iclear	116
20.16core/tn_sem.h File Reference	116
20.16.1 Detailed Description	116
20.16.2 Function Documentation	117
20.16.2.1 tn_sem_create	117
20.16.2.2 tn_sem_delete	117
20.16.2.3 tn_sem_signal	117

20.16.2.4 tn_sem_isignal	118
20.16.2.5 tn_sem_wait	118
20.16.2.6 tn_sem_wait_polling	118
20.16.2.7 tn_sem_await_polling	118
20.17 core/tn_sys.h File Reference	119
20.17.1 Detailed Description	119
20.17.2 Macro Definition Documentation	120
20.17.2.1 TN_STACK_ARR_DEF	120
20.17.2.2 _TN_BUILD_CFG_ARCH_STRUCT_FILL	121
20.17.2.3 _TN_BUILD_CFG_STRUCT_FILL	121
20.17.3 Typedef Documentation	121
20.17.3.1 TN_CBUserTaskCreate	121
20.17.3.2 TN_CBIdle	122
20.17.3.3 TN_CBStackOverflow	122
20.17.3.4 TN_CBDeadlock	122
20.17.4 Enumeration Type Documentation	122
20.17.4.1 TN_StateFlag	122
20.17.4.2 TN_Context	123
20.17.5 Function Documentation	123
20.17.5.1 tn_sys_start	123
20.17.5.2 tn_tick_int_processing	124
20.17.5.3 tn_sys_tslice_set	124
20.17.5.4 tn_sys_time_get	124
20.17.5.5 tn_callback_deadlock_set	124
20.17.5.6 tn_callback_stack_overflow_set	125
20.17.5.7 tn_sys_state_flags_get	125
20.17.5.8 tn_sys_context_get	125
20.17.5.9 tn_is_task_context	125
20.17.5.10 tn_is_isr_context	125
20.17.5.11 tn_cur_task_get	126
20.17.5.12 tn_cur_task_body_get	126
20.17.5.13 tn_callback_dyn_tick_set	126
20.18 core/tn_tasks.h File Reference	126
20.18.1 Detailed Description	126
20.18.2 Task	126
20.18.3 Task states	127
20.18.4 Creating/starting tasks	127
20.18.5 Stopping/deleting tasks	127
20.18.6 Scheduling rules	127
20.18.7 Idle task	127

20.18.8 Enumeration Type Documentation	129
20.18.8.1 TN_TaskState	129
20.18.8.2 TN_WaitReason	129
20.18.8.3 TN_TaskCreateOpt	130
20.18.8.4 TN_TaskExitOpt	130
20.18.9 Function Documentation	131
20.18.9.1 tn_task_create	131
20.18.9.2 tn_task_suspend	132
20.18.9.3 tn_task_resume	132
20.18.9.4 tn_task_sleep	133
20.18.9.5 tn_task_wakeup	133
20.18.9.6 tn_task_iwakeup	133
20.18.9.7 tn_task_activate	134
20.18.9.8 tn_task_iactivate	134
20.18.9.9 tn_task_release_wait	134
20.18.9.10 tn_task_irelease_wait	135
20.18.9.11 tn_task_exit	135
20.18.9.12 tn_task_terminate	135
20.18.9.13 tn_task_delete	136
20.18.9.14 tn_task_state_get	136
20.18.9.15 tn_task_profiler_timing_get	136
20.18.9.16 tn_task_change_priority	137
20.19 core/tn_timer.h File Reference	137
20.19.1 Detailed Description	137
20.19.2 Implementation of static timers	138
20.19.3 Typedef Documentation	139
20.19.3.1 TN_TimerFunc	139
20.19.3.2 TN_CBTickSchedule	140
20.19.3.3 TN_CBTickCntGet	140
20.19.4 Function Documentation	140
20.19.4.1 tn_timer_create	140
20.19.4.2 tn_timer_delete	141
20.19.4.3 tn_timer_start	141
20.19.4.4 tn_timer_cancel	142
20.19.4.5 tn_timer_set_func	143
20.19.4.6 tn_timer_is_active	143
20.19.4.7 tn_timer_time_left	143
20.20 tn.h File Reference	144
20.20.1 Detailed Description	144
20.21 tn_app_check.c File Reference	144

20.21.1 Detailed Description	144
20.21.2 Function Documentation	144
20.21.2.1 you_should_add_file__tn_app_check_c__to_the_project	144
20.22tn_cfg_default.h File Reference	144
20.22.1 Detailed Description	144
20.22.2 Macro Definition Documentation	146
20.22.2.1 TN_CHECK_BUILD_CFG	146
20.22.2.2 TN_PRIORITIES_CNT	146
20.22.2.3 TN_CHECK_PARAM	146
20.22.2.4 TN_DEBUG	147
20.22.2.5 TN_OLD_TNKERNEL_NAMES	147
20.22.2.6 TN_MUTEX_DEADLOCK_DETECT	147
20.22.2.7 TN_TICK_LISTS_CNT	147
20.22.2.8 TN_API_MAKE_ALIG_ARG	147
20.22.2.9 TN_PROFILER	148
20.22.2.10TN_PROFILER_WAIT_TIME	148
20.22.2.11TN_STACK_OVERFLOW_CHECK	148
20.22.2.12TN_OLD_EVENT_API	148
20.22.2.13TN_P24_SYS_IPL	149
20.22.2.14TN_P24_SYS_IPL_STR	149
Index	150

Chapter 1

TNeo overview

TNeo 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.

TNeo 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 the following architectures:

- Microchip: PIC32/PIC24/dsPIC
- ARM Cortex-M cores: Cortex-M0/M0+/M1/M3/M4/M4F

API is [changed somewhat](#), so it's not 100% compatible with TNKernel, hence the new name: TNeo.

TNeo is hosted at bitbucket: <http://bitbucket.org/dfrank/tneokernel>

Related pages:

- [Foreword](#)
- [Features](#)
- [Quick guide](#)
- [Time ticks](#)
- [Interrupts](#)
- [Building TNeo](#)
- [Architecture-specific details](#)
 - [PIC32 port details](#)
 - [PIC24/dsPIC port details](#)
 - [Cortex-M0/M0+/M3/M4/M4F port details](#)
- [Why reimplement TNKernel](#)
- [Differences from TNKernel API](#)
- [Unit tests](#)
- [Plans](#)
- [Changelog](#)
- [Thanks](#)

- [License](#)
- [Legend](#)

API reference:

- [System services](#)
- [Tasks](#)
- [Mutexes](#)
- [Semaphores](#)
- [Fixed-size memory blocks](#)
- [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 too 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 "TNeo". Refer to the [Differences from TNKernel API](#) page for details.

Together with almost totally re-writing TNKernel, I've implemented detailed [unit tests](#) for it, to make sure I didn't break anything, and of course I've found several bugs in original TNKernel 2.7: refer to the section [Bugs of TN↔Kernel 2.7](#). Unit tests are, 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 TN↔Kernel is far from perfect in my opinion, the ideas behind the implementation are generally really nice (that's why I decided to reimplement it instead of starting from scratch), and it was great entry point to the real-time kernels for me.

I would also like to thank my chiefs in the [ORION](#) company, Alexey Morozov and Alexey Gromov, for being flexible about my time.

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

Chapter 3

Features

TNeo has a complete set of common RTOS features plus some extras.

Many features are optional, so that if you don't need them you can configure the kernel as you wish and probably save memory or improve speed.

3.1 Feature list

- **Tasks**, or threads: the most common feature for which the kernel is written in the first place;
- **Mutexes**: objects for shared resources protection.
 - **Recursive mutexes**: optionally, mutexes allow nested locking;
 - **Mutex deadlock detection**: if deadlock occurs, the kernel can notify you about this problem by calling arbitrary function.
- **Semaphores**: objects for tasks synchronization;
- **Fixed-size memory blocks**: simple and deterministic memory allocator;
- **Event groups**: objects containing various event bits that tasks may set, clear and wait for;
 - **Event group connection**: extremely useful feature when you need to wait, say, for messages from multiple queues, or for other set of different events.
- **Data queues**: FIFO buffer of messages that tasks may send and receive;
- **Timers**: a tool to ask the kernel to call arbitrary function at a particular time in the future. The callback approach provides ultimate flexibility.
- **Software stack overflow check**: extremely useful feature for architectures without hardware stack pointer limit. Refer to the `TN_STACK_OVERFLOW_CHECK` option for details.
- **Dynamic tick**: if there's nothing to do, don't even bother to manage system timer tick each fixed period of time. Refer to the page [Time ticks](#) for details.
- **Profiler**: allows you to know how much time each of your tasks was actually running, get maximum consecutive running time of it, and other relevant information. Refer to the option `TN_PROFILER` and `struct TN_TaskTiming` for details.

Chapter 4

Quick guide

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

4.1 Using TNeo in your application

The easiest way is to download version archive from [downloads page](#): it contains `bin` folder with library files for various platforms. These library files were built with default configuration file (`src/tn_cfg_default.h`, see the section [Configuration file](#)).

If you use MPLABX, it is probably better to add *library project* to your main project instead; library projects reside in the `<tnéo_path>/lib_project` directory.

In either case, all you need is the following:

- Add library file for appropriate platform to your project (or probably *library project* in case of MPLABX);
- Add C include path: `<tnéo_path>/src`;
- Copy default configuration file as current configuration file: `cp <tnéo_path>/src/tn_cfg_default.h <tnéo_path>/src/tn_cfg.h` (for more information about configuration file and better ways to manage it, refer to the section [Configuration file](#))
- Add the file `<tnéo_path>/src/tn_app_check.c` to your application project (see [TN_CHECK_BUILD_CFG](#) for details on it).

That's all; you can use it in your project now. See below how to do this.

Attention

If you need to change the configuration, you can't just edit `tn_cfg.h` and keep using pre-built library file: you need to rebuild the library after editing `tn_cfg.h`. Refer to the page [Building TNeo](#) for details.

4.2 Time ticks

The kernel needs to calculate timeouts. There are two schemes available: *static tick* and *dynamic tick*. For a quick guide, it's quite enough to just read about static tick, so, for the details on it, refer to the section [Static tick](#) and then return back here.

4.3 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_ARRAY_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 *system interrupts* 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)
 - call `tn_sys_start()` providing all necessary information: pointers to stacks, their sizes and your callback functions.
- Kernel acts as follows:
 - performs all necessary housekeeping;
 - creates idle task;
 - calls your `TN_CBUserTaskCreate()` callback, in which your initial task is created with `TN_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 TNeo repository, in the `examples/basic/arch/pic32` directory.

Attention

Before trying to build examples, please read [Building TNeo](#) page carefully: you need to copy configuration file in the `tneo` directory to build it. Each example has `tn_cfg_appl.h` file, and you should either create a symbolic link to this file from `tneo/src/tn_cfg.h` or just copy this file as `tneo/src/tn_cfg.h`.

```
/**
 * TNeo 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            // CLK0 Enable
#pragma config POSCMOD  = HS             // Primary Oscillator
#pragma config IESO     = OFF            // Internal/External Switch-over
#pragma config FSOCEN   = 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-vectored 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_a_body,
        TASK_A_PRIORITY,
        task_a_stack,
        TASK_A_STK_SIZE,
        NULL,
        TN_TASK_CREATE_OPT_START
    );
}

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 (;;) ;
}

```

4.4 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.

Attention

Round-robin is not supported in [Dynamic tick](#) mode.

Chapter 5

Time ticks

The kernel needs to calculate timeouts.

There are two schemes available: *static tick* and *dynamic tick*.

5.1 Static tick

Static tick is the easiest way to implement timeouts: there should be just some kind of hardware timer that generates interrupts periodically. 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();
}
```

But for some applications that spend a lot of time doing nothing this could be far from perfect: instead of being constantly in the power-saving mode while there's nothing to do, the CPU needs to wake up regularly. So, *dynamic tick* scheme was implemented:

5.2 Dynamic tick

The general idea is that there should be no useless calls to `tn_tick_int_processing()`. If the kernel needs to wake up after 100 system ticks, then, `tn_tick_int_processing()` should be called exactly after 100 periods of system tick (but external asynchronous events still can happen and re-schedule that, of course).

To this end, the kernel should be able to communicate with the application:

- To schedule next call to `tn_tick_int_processing()` after N ticks;
- To ask what time is now (i.e. get current system ticks count)

To use dynamic tick, turn the option `TN_DYNAMIC_TICK` to 1.

Then, a couple of callback prototypes becomes available:

- `TN_CBTickSchedule;`
- `TN_CBTickCntGet.`

And you must provide these callbacks to `tn_callback_dyn_tick_set()` **before** starting the system (i.e. before calling `tn_sys_start()`)

Attention

In dynamic tick mode, `round-robin` is not yet supported.

Chapter 6

Interrupts

6.1 Interrupt stack

TNeo 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](#).

The way a separate interrupt stack is implemented is architecture-specific, as well as the way to define an ISR: some platforms require kernel-provided macro for that, some don't. Refer to the section for particular architecture:

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

6.2 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 7

Building TNeo

Some notes on building the project.

Note: you don't *have* to build TNeo to use it. If you want to just use pre-built library (with default configuration), refer to the section [Using TNeo in your application](#).

7.1 Configuration file

TNeo 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 TNeo source directory, like this:

```
$ cd /path/to/tnéo/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.

7.2 Makefile or library projects

If you need to build TNeo with some non-default configuration, the easiest way is to use ready-made Makefile or library project.

7.2.1 Makefile

It is tested only in Unix-like environment, so that you can't use makefile to build the kernel with Keil Realview or IAR. For Keil Realview or IAR, use library project (see the section below).

There are two makefiles available: `Makefile-all-arch` and `Makefile`.

The first one is used to build all possible targets at once, so it is more for the kernel developer than for kernel user. The second one is used to build the kernel for some particular architecture, and it takes two params: `TN_ARCH` and `TN_COMPILER`.

Valid values for `TN_ARCH` are:

- `cortex_m0` - for Cortex-M0 architecture,
- `cortex_m0plus` - for Cortex-M0+ architecture,
- `cortex_m1` - for Cortex-M1 architecture,
- `cortex_m3` - for Cortex-M3 architecture,
- `cortex_m4` - for Cortex-M4 architecture,
- `cortex_m4f` - for Cortex-M4F architecture,
- `pic32mx` - for PIC32MX architecture,
- `pic24_dspic_noeds` - for PIC24/dsPIC architecture without EDS (Extended Data Space),
- `pic24_dspic_eds` - for PIC24/dsPIC architecture with EDS.

Valid values for `TN_COMPILER` depend on architecture. For Cortex-M series, they are:

- `arm-none-eabi-gcc` (you need [GNU ARM Embedded toolchain](#))
- `clang` (you need [LLVM clang](#))

For PIC32, just one value is valid:

- `xc32` (you need [Microchip XC32 compiler](#))

For PIC24/dsPIC, just one value is valid:

- `xc16` (you need [Microchip XC16 compiler](#))

Example invocation (from the TNeo's root directory) :

```
$ make TN_ARCH=cortex_m3 TN_COMPILER=arm-none-eabi-gcc
```

As a result, there will be archive library file `bin/cortex_m3/arm-none-eabi-gcc/tneo_cortex_m3_↵_arm-none-eabi-gcc.a`

7.2.2 Library project

In the root of TNeo repository, there is a directory `lib_project` which contains ready-made projects for various platforms. You may use it for building library, and then use resulting library file in your project.

For MPLABX projects, there are *library projects*, so that you even don't need to build a library: just add this *library project* to your main project, and MPLABX will do all the work for you. You can change `tn_cfg.h` file "on-the-fly" then. Other IDEs don't offer such a luxuries, so you need to build library file as a separate step.

7.3 Building manually

If you want to create library project yourself (say, in some different IDE, or anything), or if you want to build TNeo as a direct part of your project, there are some generic requirements (there might be additional architecture-dependent requirements, see links below):

- **Core sources:** add all `.c` files from `src/core` directory to the project.
- **C99:** TNeo uses some features of C99, such as `static inline` functions and variable declarations not at the start of a compound statement. So, C99 is a requirement.

- **C Include directories** (relative to the root of the repository) :
 - src
 - src/core
 - src/core/internal
 - src/arch
- **Assembler preprocessor Include directories** (relative to the root of the repository) :
 - src
 - src/core
- **.S files preprocessed by C preprocessor**: This is probably more arch-dependent requirement than a generic one, but actually .S files for all supported architectures need to be preprocessed, so it is specified here. On most platforms, it works "out-of-the-box", on some others, you need to perform additional steps for it: in these cases, necessary steps explained in the "building" section for the appropriate architecture, see links below.
- **Isolate each function in a section** Not a requirement, but recommendation: for embedded designs, it is usually a good idea to isolate each function in a section, so that in your application you can set linker option like "remove unused sections", and save notable amount of flash memory.

For arch-dependent information on building TNeo, please refer to the appropriate section:

- [Building for PIC24/dsPIC](#)
- [Building for PIC32](#)
- [Building for Cortex-M0/M1/M3/M4/M4F](#)

Chapter 8

Architecture-specific details

Architecture-specific details

8.1 PIC32 port details

8.1.1 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 tneo is built as a separate library (which is typically the case), 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 TNeo 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.

8.1.2 Interrupts

For generic information about interrupts in TNeo, 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**.

8.1.3 Building

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

MPLABX project for PIC32 port resides in the `lib_project/pic32/tneo_pic32.X` directory. This is a *library project* in terms of MPLABX, so if you use MPLABX you can easily add it to your application project by right-clicking `Libraries -> Add Library Project ...`. Alternatively, of course you can just build it and use resulting `tneo_pic32.X.a` file in whatever way you like.

If you want to build TNeo manually, refer to the section [Building manually](#) for generic notes about it, and there is a couple of arch-dependent sources you need to add to the project:

- `src/arch/pic32/tn_arch_pic32.c`
- `src/arch/pic32/tn_arch_pic32mx_xc32.S`

Attention

There is one more file: `tn_arch_pic32_int_vec1.S`, which should be included **in your application project** to make things work. It is needed to dispatch vector1 (Core Software Interrupt 0) correctly.

8.2 PIC24/dsPIC port details

8.2.1 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.

8.2.2 Interrupts

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

PIC24/dsPIC TNeo port supports nested interrupts. It allows to specify the range of *system interrupt priorities*. Refer to the subsection [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_IPL`). Use standard way to define it. If `TN_CHECK_PARAM` is on, kernel checks it: if you violate this rule, debugger will be halted by the kernel when entering ISR. In release build, CPU is just reset.

8.2.3 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 TNeo 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.

8.2.4 Building

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

MPLABX project for PIC24/dsPIC port resides in the `lib_project/pic24_dspic/tneo_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`

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

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.

If you want to build TNeo manually, refer to the section [Building manually](#) for generic notes about it, and additionally you should add arch-dependent sources: all `.c` and `.S` files from `src/arch/pic24_dspic`

8.3 Cortex-M0/M0+/M3/M4/M4F port details

8.3.1 Context switch

The context switch is implemented in a standard for Cortex-M CPUs way: the PendSV exception. SVC exception is used for `_tn_arch_context_switch_now_nosave()`. These two exceptions are configured by the kernel to the lowest priority.

8.3.2 Interrupts

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

Cortex-M port has *system interrupts* only, there are no *user interrupts*.

Interrupts use separate interrupt stack, i.e. MSP (Main Stack Pointer). Tasks use PSP (Process Stack Pointer).

There are no constraints on ISRs: no special macros for ISR definition, or whatever. This is because Cortex-↔M processors are designed with OS applications in mind, so a number of features are available to make OS implementation easier and make OS operations more efficient.

8.3.3 Building

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

There are many environments for building for Cortex-M CPUs (Keil, Eclipse, CoCoX), all available projects reside in `lib_project/cortex_m` directory. They usually are pretty enough if you want to just build the kernel with non-default configuration.

If, however, you want to build it not using provided project, refer to the section [Building manually](#) for generic notes about it, and additionally you should add arch-dependent sources: all `.c` and `.S` files from `src/arch/cortex-m`.

There are some additional tips depending on the build environment:

Keil 5, ARMCC compiler

To satisfy [building requirements](#), a couple of actions needed:

- C99 is off by default. In project options, C/C++ tab, check "C99 Mode" checkbox.
- Assembler files (`.S`) aren't preprocessed by default, so, in project options, Asm tab, "Misc Controls" field, type the following: `--cpreproc`

Keil 5, GCC compiler

Unfortunately, when GCC toolchain is used from Keil uVision IDE, for `.S` files it calls `arm-none-eabi-as`, which does not call C preprocessor.

Instead, `arm-none-eabi-gcc` should be used, but unfortunately I was unable to make Keil uVision issue `arm-none-eabi-gcc` for `.S` files, the only way to use GCC toolchain in Keil uVision that I'm aware of is to preprocess the file manually, like that:

```
cpp -P -undef tn_arch_cortex_m.S \
-D __GNUC__ -D __ARM_ARCH -D __ARM_ARCH_7M__ \
-I ../../ -I ../../core \
> tn_arch_cortex_m3_gcc.s
```

(this example is for Cortex-M3, you may check the file `tn_arch_detect.h` to see what should you define instead of `__ARM_ARCH_7M__` for other cores)

And then, add the output file `tn_arch_cortex_m3_gcc.s` to the project instead of `tn_arch_cortex_m.S`

Chapter 9

Why reimplement TNKernel

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

9.1 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).

9.2 Examples of poor implementation

9.2.1 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.

9.2.2 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 TNeo.

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](#)).

9.2.3 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;
}
```

9.2.4 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`.

9.3 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_fmem_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 TNeo 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 10

Differences from TNKernel API

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

10.1 Incompatible API changes

10.1.1 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.

10.1.2 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 ... */);
```

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

10.1.3 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 TNeo 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.

10.1.4 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, TNeo returns `TN_RC_WPARAM` in these cases. User must provide `start_addr` and `block_size` that are properly aligned.

TNeo 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
}
```

10.1.5 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, TNeo returns `TN_RC_WSTATE` in these cases.

10.1.6 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.

10.1.7 Return code of `tn_task_sleep()`

In original TNKernel, `tn_task_sleep()` always returns `TERR_NO_ERR`, independently of what actually happened. In TNeo, 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()`;

10.1.8 Events API is changed almost completely

Note: for old TNKernel projects, there is a compatibility mode, see `TN_OLD_EVENT_API`.

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 TNeo, 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. Instead, you may set the flag `TN_EVENTGRP_WMODE_AUTOCLR` when task is going to wait for some event bit(s), and then these event bit(s) will be atomically cleared automatically when task successfully finishes waiting for these bits.

TNeo also offers a very useful feature: connecting an event group to other kernel objects. Refer to the section [Connecting an event group to other system objects](#).

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

10.1.9 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 TNeo system functions allows timeout to be zero: in this case, function doesn't wait.

10.2 New features

Well, I'm tired of maintaining this additional list of features, so I just say that there is a lot of new features: timers, event group connection, stack overflow check, recursive mutexes, mutex deadlock detection, profiler, dynamic tick, etc.

Refer to the generic [feature list](#).

10.3 Compatible API changes

10.3.1 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_`?

TNeo 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.

10.3.2 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.

10.3.3 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_fmem_create()` for usage example.

10.3.4 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_fmem_get_polling()` / `tn_fmem_get_ipolling()` (notice the `i` letter before `polling`)?
- All the system service names follow the naming scheme `tn_<noun>_<verb>[_<adjustment>]()`, but the `tn_start_system()` is special, for some strange reason. To make it consistent, it should be named `tn_system_start()` or `tn_sys_start()`;
- A lot of macros don't have `TN_` prefix;

- etc

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

10.3.5 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`.

10.4 Changes that do not affect API directly

10.4.1 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 11

Unit tests

Brief information on the implementation of unit tests

11.1 Tested CPUs

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

- PIC32MX440F512H
- PIC24FJ256GB106

11.2 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, TWORKEER_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, TWORKEER_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_tnttest/appl_tnttest_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_tnttest/appl_tnttest_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_tnttest/appl_tnttest_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.

11.3 Get unit-tests

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

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

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

There is an "environment" repository, which contains tests and all the necessary library subrepos: http://hg.dfrank.ru/tntest/_env

You can clone it as follows:

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

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

Chapter 12

Plans

No specific plans at the moment.

Chapter 13

Changelog

TNeo changelog

13.1 v1.05

Release date: 2015-01-02.

- Cortex-M0/M0+/M1/M3/M4/M4F architectures are now supported.
 - The following compilers are tested:
 - * ARMCC (Keil RealView)
 - * GCC
 - Should work but not tested carefully:
 - * clang
 - * IAR
- Software task stack overflow check (optionally), see [TN_STACK_OVERFLOW_CHECK](#) for details.
- Dynamic tick, or *tickless* (optionally): refer to the page [Time ticks](#) for details.
- Profiler (optionally): allows to see how much time task was running, how much time it was waiting and for what it was waiting, and so on. Refer to the documentation of `struct TN_TaskTiming` for details.
- Old TNKernel events compatibility mode, see [TN_OLD_EVENT_API](#) for details.
- Event groups: added [TN_EVENTGRP_WMODE_AUTOCLR](#) flag which allows to clear event bits atomically when task successfully finishes waiting for these event bits.
- PIC24/dsPIC: little optimization: ffs (find-first-set bit) is implemented in an efficient PIC24/dsPIC-specific way, so finding next task to run now works a bit faster.
- Added run-time check which ensures that build-time options for the kernel match ones for the application. For details, refer to the option [TN_CHECK_BUILD_CFG](#). **Note:** in your existing project that uses TNeo as a separate library, you need either:
 - Include the file `<tno_path>/src/tn_app_check.c` to the application project (recommended);
 - In your `tn_cfg.h` file, set [TN_CHECK_BUILD_CFG](#) to 0 and rebuild the kernel with the new configuration (not recommended).

But if you build TNeo together with the application, this option is useless, so then just set [TN_CHECK_BUILD_CFG](#) to 0.

- MPLABX projects for PIC32 and PIC24/dsPIC moved to `lib_project` directory. If you use these library projects from the repository directly in your application, you need to modify path to the library project in your application project.
- The project's name is shortened to **TNeo**.

13.2 v1.04

Release date: 2014-11-04.

- Added PIC24/dsPIC support, refer to the page [PIC24/dsPIC port details](#);
- PIC32: Core Software Interrupt is now handled by the kernel completely, application shouldn't set it up anymore. Refer to the page [PIC32 port 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 `tneo.X` to `tneo_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.

13.3 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 TNeo [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.

13.4 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 TNeo uses internal system tick count for timers, and modifying system tick counter by user is a *really* bad idea.

13.5 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`.

13.6 v1.0

Release date: 2014-10-01.

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

Chapter 14

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 used to use for a long time;
- **StarLine company** - for being a sponsor of kernel port for Cortex-M architecture as well as a couple of other [features](#): profiler, dynamic tick.
- **Alexey Morozov** and **Alexey Gromov**, my chiefs in the [ORION](#) company, for being flexible about my time;
- **Robert White** - for nice ideas and participation.

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

Chapter 15

License

TNeo: real-time kernel initially based on TNKernel

- TNKernel: copyright © 2004, 2013 Yuri Tiomkin.
- PIC32-specific routines: copyright © 2013, 2014 Anders Montonen.
- TNeo: copyright © 2014 Dmitry Frank.

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

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





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

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

Chapter 16

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 17

Data Structure Index

17.1 Data Structures

Here are the data structures with brief descriptions:

_TN_BuildCfg	Structure with build-time configurations values; it is needed for run-time check which ensures that build-time options for the kernel match ones for the application	59
_TN_TaskProfiler	Internal kernel structure for profiling data of task	60
TN_DQueue	Structure representing data queue object	61
TN_DQueueTaskWait	DQueue-specific fields related to waiting task, to be included in struct TN_Task	61
TN_EGrpLink	A link to event group: used when event group can be connected to some kernel object, such as queue	62
TN_EGrpTaskWait	EventGrp-specific fields related to waiting task, to be included in struct TN_Task	62
TN_EventGrp	Event group	63
TN_FMem	Fixed memory blocks pool	63
TN_FMemTaskWait	FMem-specific fields related to waiting task, to be included in struct TN_Task	64
TN_ListItem	Circular doubly linked list item, for internal kernel usage	64
TN_Mutex	Mutex	65
TN_Sem	Semaphore	65
TN_Task	Task	66
TN_TaskTiming	Timing structure that is managed by profiler and can be read by tn_task_profiler_get_timing_get() function	68
TN_Timer	Timer	69

Chapter 18

File Index

18.1 File List

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

tn.h	The main kernel header file that should be included by user application; it merely includes subsystem-specific kernel headers	144
tn_app_check.c	If TN_CHECK_BUILD_CFG option is non-zero, this file needs to be included in the application project	144
tn_cfg_default.h	TNeo default configuration file, to be copied as tn_cfg.h	144
arch/tn_arch.h	Architecture-dependent routines declaration	84
arch/cortex_m/tn_arch_cortex_m.h	Cortex-M0/M0+/M3/M4/M4F architecture-dependent routines	71
arch/example/tn_arch_example.h	Example of architecture-dependent routines	71
arch/pic24_dspic/tn_arch_pic24.h	PIC24/dsPIC architecture-dependent routines	75
arch/pic24_dspic/tn_arch_pic24_bfa.h	Atomic bit-field access macros for PIC24/dsPIC	76
arch/pic32/tn_arch_pic32.h	PIC32 architecture-dependent routines	79
arch/pic32/tn_arch_pic32_bfa.h	Atomic bit-field access macros for PIC24/dsPIC	81
core/tn_cfg_dispatch.h	Dispatch configuration: set predefined options, include user-provided cfg file as well as default cfg file	88
core/tn_common.h	Definitions used through the whole kernel	89
core/tn_dqueue.h	A data queue is a FIFO that stores pointer (of type <code>void *</code>) in each cell, called (in uITRON style) a data element	93
core/tn_eventgrp.h	Event group	98
core/tn_fmem.h	Fixed memory blocks pool	103
core/tn_list.h	Circular doubly linked list, for internal kernel usage	106
core/tn_mutex.h	A mutex is an object used to protect shared resources	107

core/tn_oldsymbols.h	Compatibility layer for old projects that use old TNKernel names; usage of them in new projects is discouraged	110
core/tn_sem.h	A semaphore: an object to provide signaling mechanism	116
core/tn_sys.h	Kernel system routines: system start, tick processing, time slice managing	119
core/tn_tasks.h	126
core/tn_timer.h	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	137

Chapter 19

Data Structure Documentation

19.1 `_TN_BuildCfg` Struct Reference

19.1.1 Detailed Description

Structure with build-time configurations values; it is needed for run-time check which ensures that build-time options for the kernel match ones for the application.

See [TN_CHECK_BUILD_CFG](#) for details.

Definition at line 158 of file `tn_sys.h`.

Data Fields

- unsigned `priorities_cnt`: 7
Value of `TN_PRIORITIES_CNT`
- unsigned `check_param`: 1
Value of `TN_CHECK_PARAM`
- unsigned `debug`: 1
Value of `TN_DEBUG`
- unsigned `use_mutexes`: 1
Value of `TN_USE_MUTEXES`
- unsigned `mutex_rec`: 1
Value of `TN_MUTEX_REC`
- unsigned `mutex_deadlock_detect`: 1
Value of `TN_MUTEX_DEADLOCK_DETECT`
- unsigned `tick_lists_cnt_minus_one`: 8
Value of `TN_TICK_LISTS_CNT` minus one.
- unsigned `api_make_alig_arg`: 2
Value of `TN_API_MAKE_ALIG_ARG`
- unsigned `profiler`: 1
Value of `TN_PROFILER`
- unsigned `profiler_wait_time`: 1
Value of `TN_PROFILER_WAIT_TIME`
- unsigned `stack_overflow_check`: 1
Value of `TN_STACK_OVERFLOW_CHECK`
- unsigned `dynamic_tick`: 1
Value of `TN_DYNAMIC_TICK`
- unsigned `old_events_api`: 1

```

    Value of TN\_OLD\_EVENT\_API
    • union {
        TN\_UWord dummy
        On some architectures, we don't have any arch-dependent build-time options, but we need this "dummy" value to avoid er
        struct {
            unsigned p24\_sys\_ipl: 3
                Value of TN\_P24\_SYS\_IPL
        } p24
            PIC24/dsPIC-dependent values.
        } arch

        Architecture-dependent values.

```

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

- [core/tn_sys.h](#)

19.2 _TN_TaskProfiler Struct Reference

19.2.1 Detailed Description

Internal kernel structure for profiling data of task.

Available if only [TN_PROFILER](#) option is non-zero.

Definition at line 302 of file [tn_tasks.h](#).

Data Fields

- [TN_TickCnt](#) [last_tick_cnt](#)
Tick count of when the task got running or non-running last time.
- enum [TN_WaitReason](#) [last_wait_reason](#)
Available if only [TN_PROFILER_WAIT_TIME](#) option is non-zero.
- int [is_running](#)
For internal profiler self-check only: indicates whether task is running or not.
- struct [TN_TaskTiming](#) [timing](#)
Main timing structure managed by profiler.

19.2.2 Field Documentation

19.2.2.1 enum [TN_WaitReason](#) [_TN_TaskProfiler::last_wait_reason](#)

Available if only [TN_PROFILER_WAIT_TIME](#) option is non-zero.

Value of `task->task_wait_reason` when task got non-running last time.

Definition at line 311 of file [tn_tasks.h](#).

19.2.2.2 int [_TN_TaskProfiler::is_running](#)

For internal profiler self-check only: indicates whether task is running or not.

Available if only [TN_DEBUG](#) is non-zero.

Definition at line 318 of file [tn_tasks.h](#).

19.2.2.3 struct TN_TaskTiming _TN_TaskProfiler::timing

Main timing structure managed by profiler.

Contents of this structure can be read by `tn_task_profiler_timing_get()` function.

Definition at line 323 of file `tn_tasks.h`.

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

- `core/tn_tasks.h`

19.3 TN_DQueue Struct Reference

19.3.1 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`

19.4 TN_DQueueTaskWait Struct Reference

19.4.1 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](#)

19.5 TN_EGrpLink Struct Reference

19.5.1 Detailed Description

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

Definition at line 244 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](#)

19.6 TN_EGrpTaskWait Struct Reference

19.6.1 Detailed Description

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

Definition at line 228 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](#)

19.7 TN_EventGrp Struct Reference

19.7.1 Detailed Description

Event group.

Definition at line 210 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
- enum [TN_EGrpAttr attr](#)
Attributes that are given to that events group, available if only [TN_OLD_EVENT_API](#) option is non-zero.

19.7.2 Field Documentation

19.7.2.1 enum TN_EGrpAttr TN_EventGrp::attr

Attributes that are given to that events group, available if only [TN_OLD_EVENT_API](#) option is non-zero.

Definition at line 219 of file tn_eventgrp.h.

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

- core/[tn_eventgrp.h](#)

19.8 TN_FMem Struct Reference

19.8.1 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

19.8.2 Field Documentation

19.8.2.1 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_ALIGN_SIZE\(\)](#) for that.

See also

[TN_MAKE_ALIGN_SIZE\(\)](#)

Definition at line 88 of file [tn_fmem.h](#).

19.8.2.2 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_fmem.h](#).

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

- [core/tn_fmem.h](#)

19.9 TN_FMemTaskWait Struct Reference

19.9.1 Detailed Description

FMem-specific fields related to waiting task, to be included in struct [TN_Task](#).

Definition at line 112 of file [tn_fmem.h](#).

Data Fields

- void * [data_elem](#)
if task tries to receive data from memory pool, and there's no more free blocks in the pool, location to store pointer is saved in this field

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

- [core/tn_fmem.h](#)

19.10 TN_ListItem Struct Reference

19.10.1 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](#)

19.11 TN_Mutex Struct Reference

19.11.1 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](#)

19.12 TN_Sem Struct Reference

19.12.1 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](#)

19.13 TN_Task Struct Reference

19.13.1 Detailed Description

Task.

Definition at line 330 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_fmem.h](#)*
 } [subsys_wait](#)
 - subsystem-specific fields that are used while task waits for something.*
- const char * [name](#)
 - Task name for debug purposes, user may want to set it by hand.*
- struct [_TN_TaskProfiler profiler](#)
 - Profiler data, available if only [TN_PROFILER](#) is non-zero.*
- 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.*

19.13.2 Field Documentation

19.13.2.1 [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 334 of file `tn_tasks.h`.

19.13.2.2 [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 361 of file `tn_tasks.h`.

19.13.2.3 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.

19.13.2.4 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 434 of file `tn_tasks.h`.

19.13.2.5 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 440 of file `tn_tasks.h`.

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

- [core/tn_tasks.h](#)

19.14 TN_TaskTiming Struct Reference

19.14.1 Detailed Description

Timing structure that is managed by profiler and can be read by `tn_task_profiler_timing_get()` function.

This structure is contained in each `struct TN_Task` structure.

Available if only `TN_PROFILER` option is non-zero, also depends on `TN_PROFILER_WAIT_TIME`.

Definition at line 254 of file `tn_tasks.h`.

Data Fields

- unsigned long long `total_run_time`
Total time when task was running.
- unsigned long long `got_running_cnt`
How many times task got running.
- unsigned long `max_consecutive_run_time`
Maximum consecutive time task was running.
- unsigned long long `total_wait_time` [`TN_WAIT_REASONS_CNT`]
Available if only `TN_PROFILER_WAIT_TIME` option is non-zero.
- unsigned long `max_consecutive_wait_time` [`TN_WAIT_REASONS_CNT`]
Available if only `TN_PROFILER_WAIT_TIME` option is non-zero.

19.14.2 Field Documentation

19.14.2.1 unsigned long long TN_TaskTiming::total_run_time

Total time when task was running.

Attention

This is NOT the time that task was in [RUNNABLE](#) state: if task A is preempted by high-priority task B, task A is not running, but is still in the [RUNNABLE](#) state. This counter represents the time task was actually **running**.

Definition at line 263 of file `tn_tasks.h`.

19.14.2.2 unsigned long long TN_TaskTiming::got_running_cnt

How many times task got running.

It is useful to find an average value of consecutive running time: $(total_run_time / got_running_cnt)$

Definition at line 267 of file `tn_tasks.h`.

19.14.2.3 unsigned long long TN_TaskTiming::total_wait_time[TN_WAIT_REASONS_CNT]

Available if only [TN_PROFILER_WAIT_TIME](#) option is non-zero.

Total time when task was not running; time is broken down by reasons of waiting.

For example, to get the time task was waiting for mutexes with priority inheritance protocol, use: `total_wait_time[TN_WAIT_REASON_MUTEX_I]`

To get the time task was runnable but preempted by another task, use: `total_wait_time[TN_WAIT_REASON_NONE]`

Definition at line 285 of file `tn_tasks.h`.

19.14.2.4 unsigned long TN_TaskTiming::max_consecutive_wait_time[TN_WAIT_REASONS_CNT]

Available if only [TN_PROFILER_WAIT_TIME](#) option is non-zero.

Maximum consecutive time task was not running; time is broken down by reasons of waiting.

See also

[total_wait_time](#)

Definition at line 293 of file `tn_tasks.h`.

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

- [core/tn_tasks.h](#)

19.15 TN_Timer Struct Reference

19.15.1 Detailed Description

Timer.

Definition at line 205 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_TickCnt start_tick_cnt](#)
Available if only [TN_DYNAMIC_TICK](#) is **set**.
- [TN_TickCnt timeout](#)
Available if only [TN_DYNAMIC_TICK](#) is **set**.
- [TN_TickCnt timeout_cur](#)
Available if only [TN_DYNAMIC_TICK](#) is **not set**.
- enum [TN_ObjId id_timer](#)
id for object validity verification

19.15.2 Field Documentation

19.15.2.1 [TN_TickCnt](#) [TN_Timer::start_tick_cnt](#)

Available if only [TN_DYNAMIC_TICK](#) is **set**.

Tick count value when timer was started

Definition at line 221 of file [tn_timer.h](#).

19.15.2.2 [TN_TickCnt](#) [TN_Timer::timeout](#)

Available if only [TN_DYNAMIC_TICK](#) is **set**.

Timeout value (it is set just once, and stays unchanged until timer is expired, cancelled or restarted)

Definition at line 227 of file [tn_timer.h](#).

19.15.2.3 [TN_TickCnt](#) [TN_Timer::timeout_cur](#)

Available if only [TN_DYNAMIC_TICK](#) is **not set**.

Current (left) timeout value

Definition at line 235 of file [tn_timer.h](#).

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

- [core/tn_timer.h](#)

Chapter 20

File Documentation

20.1 arch/cortex_m/tn_arch_cortex_m.h File Reference

20.1.1 Detailed Description

Cortex-M0/M0+/M3/M4/M4F architecture-dependent routines.

Definition in file [tn_arch_cortex_m.h](#).

20.2 arch/example/tn_arch_example.h File Reference

20.2.1 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_TickCnt)0xFFFFFFFF`
Value for infinite waiting, usually matches `ULONG_MAX`, because `TN_TickCnt` 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`.
- `#define _TN_INLINE` `inline`
If compiler does not conform to c99 standard, there's no inline keyword.
- `#define _TN_VOLATILE_WORKAROUND` `/* nothing */`
Sometimes compilers are buggy in high-optimization modes, and these bugs are often could be worked around by adding the `volatile` keyword.

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.

20.2.2 Macro Definition Documentation

20.2.2.1 `#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`.

20.2.2.2 `#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 TNeo bugs as well as illegal kernel usage, e.g. sleeping in the idle task callback or build-time configuration mismatch (see `TN_CHECK_BUILD_CFG` for details on the last one)

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

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

20.2.2.3 #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 77 of file tn_arch_example.h.

20.2.2.4 #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 88 of file tn_arch_example.h.

20.2.2.5 #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 120 of file tn_arch_example.h.

20.2.2.6 #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 143 of file tn_arch_example.h.

20.2.2.7 #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 152 of file tn_arch_example.h.

20.2.2.8 `#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 164 of file `tn_arch_example.h`.

20.2.2.9 `#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 176 of file `tn_arch_example.h`.

20.2.2.10 `#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 185 of file `tn_arch_example.h`.

20.2.2.11 `#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 194 of file `tn_arch_example.h`.

20.2.2.12 `#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 213 of file `tn_arch_example.h`.

20.2.2.13 `#define _TN_INLINE inline`

If compiler does not conform to c99 standard, there's no inline keyword.

So, there's a special macro for that.

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

20.2.2.14 `#define _TN_VOLATILE_WORKAROUND /* nothing */`

Sometimes compilers are buggy in high-optimization modes, and these bugs are often could be worked around by adding the `volatile` keyword.

It is compiler-dependent, so, there's a special macro for that.

Definition at line 227 of file `tn_arch_example.h`.

20.2.3 Typedef Documentation

20.2.3.1 `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 105 of file `tn_arch_example.h`.

20.2.3.2 `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 112 of file `tn_arch_example.h`.

20.3 arch/pic24_dspic/tn_arch_pic24.h File Reference

20.3.1 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.

20.3.2 Macro Definition Documentation

20.3.2.1 `#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(_TlInterrupt, auto_psv)
{
    //-- clear interrupt flag
    IFS0bits.TlIF = 0;

    //-- do something useful
}
```

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

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

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

Definition at line 456 of file `tn_arch_pic24.h`.

20.4 arch/pic24_dspic/tn_arch_pic24_bfa.h File Reference

20.4.1 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 0BBBB`
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.

20.4.2 Macro Definition Documentation

20.4.2.1 `#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`.

20.4.2.2 #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.

20.4.2.3 #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.

20.4.2.4 #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.

20.4.2.5 #define TN_BFA_RD 0xBBBB

Command for [TN_BFA\(\)](#) macro: Read bit field; . . . macro param ignored.

Definition at line 92 of file tn_arch_pic24_bfa.h.

20.4.2.6 #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">• TN_BFA_WR - write bit field• TN_BFA_RD - read bit field• TN_BFA_SET - set bits by mask• TN_BFA_CLR - clear bits by mask• TN_BFA_INV - invert bits by mask
-------------	---

<i>reg_name</i>	register name (PORTA, CMCON, ...).
<i>field_name</i>	structure field name
...	used if only <code>comm != TN_BFA_RD</code> . Meaning depends on the <code>comm</code> , see comments for specific command: <code>TN_BFA_WR</code> , etc.

Usage examples:

```
int a = 0x02;

//-- Set third bit of the INT0IP field in the IPC0 register:
//  IPC0bits.INT0IP |= (1 << 2);
TN_BFA(TN_BFA_SET, IPC0, INT0IP, (1 << 2));

//-- Clear second bit of the INT0IP field in the IPC0 register:
//  IPC0bits.INT0IP &= ~(1 << 1);
TN_BFA(TN_BFA_CLR, IPC0, INT0IP, (1 << 1));

//-- Invert two less-significant bits of the INT0IP field
//  in the IPC0 register:
//  IPC0bits.INT0IP ^= 0x03;
TN_BFA(TN_BFA_INV, IPC0, INT0IP, 0x03);

//-- Write value 0x05 to the INT0IP field of the IPC0 register:
//  IPC0bits.INT0IP = 0x05;
TN_BFA(TN_BFA_WR, IPC0, INT0IP, 0x05);

//-- Write value of the variable a to the INT0IP field of the IPC0
//  register:
//  IPC0bits.INT0IP = a;
TN_BFA(TN_BFA_WR, IPC0, INT0IP, a);

//-- Read the value that is stored in the INT0IP field of the IPC0
//  register, to the int variable a:
//  int a = IPC0bits.INT0IP;
a = TN_BFA(TN_BFA_RD, IPC0, INT0IP);
```

Definition at line 154 of file `tn_arch_pic24_bfa.h`.

20.4.2.7 `#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"> • <code>TN_BFA_WR</code> - write bit field • <code>TN_BFA_RD</code> - read bit field • <code>TN_BFA_SET</code> - set bits by mask • <code>TN_BFA_CLR</code> - clear bits by mask • <code>TN_BFA_INV</code> - invert bits by mask
-------------	--

<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 <code>comm != TN_BFA_RD</code> . Meaning depends on the <code>comm</code> , see comments for specific command: <code>TN_BFA_WR</code> , 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`.

20.5 arch/pic32/tn_arch_pic32.h File Reference

20.5.1 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.

20.5.2 Macro Definition Documentation

20.5.2.1 `#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

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

Definition at line 329 of file `tn_arch_pic32.h`.

20.5.2.2 `#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

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

Definition at line 478 of file `tn_arch_pic32.h`.

20.5.3 Variable Documentation

20.5.3.1 `volatile int tn_p32_int_nest_count`

current interrupt nesting count.

Used by macros `tn_p32_soft_isr()` and `tn_p32_srs_isr()`.

20.5.3.2 `void* tn_p32_user_sp`

saved task stack pointer.

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

20.5.3.3 `void* tn_p32_int_sp`

saved ISR stack pointer.

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

20.6 arch/pic32/tn_arch_pic32_bfa.h File Reference

20.6.1 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_pic32_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 0xBBBB`
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.

20.6.2 Macro Definition Documentation

20.6.2.1 `#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_pic32_bfa.h](#).

20.6.2.2 `#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_pic32_bfa.h](#).

20.6.2.3 `#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_pic32_bfa.h](#).

20.6.2.4 #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_pic32_bfa.h`.

20.6.2.5 #define TN_BFA_RD 0xBBBB

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

Definition at line 92 of file `tn_arch_pic32_bfa.h`.

20.6.2.6 #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"> • <code>TN_BFA_WR</code> - write bit field • <code>TN_BFA_RD</code> - read bit field • <code>TN_BFA_SET</code> - set bits by mask • <code>TN_BFA_CLR</code> - clear bits by mask • <code>TN_BFA_INV</code> - invert bits by mask
<i>reg_name</i>	register name (PORTA, CMCON, ...).
<i>field_name</i>	structure field name
...	used if only <code>comm</code> != <code>TN_BFA_RD</code> . Meaning depends on the <code>comm</code> , see comments for specific command: <code>TN_BFA_WR</code> , etc.

Usage examples:

```
int a = 0x02;

//-- Set third bit of the INT0IP field in the IPC0 register:
//  IPC0bits.INT0IP |= (1 << 2);
TN_BFA(TN_BFA_SET, IPC0, INT0IP, (1 << 2));

//-- Clear second bit of the INT0IP field in the IPC0 register:
//  IPC0bits.INT0IP &= ~(1 << 1);
TN_BFA(TN_BFA_CLR, IPC0, INT0IP, (1 << 1));

//-- Invert two less-significant bits of the INT0IP field
//  in the IPC0 register:
//  IPC0bits.INT0IP ^= 0x03;
TN_BFA(TN_BFA_INV, IPC0, INT0IP, 0x03);

//-- Write value 0x05 to the INT0IP field of the IPC0 register:
//  IPC0bits.INT0IP = 0x05;
TN_BFA(TN_BFA_WR, IPC0, INT0IP, 0x05);

//-- Write value of the variable a to the INT0IP field of the IPC0
//  register:
//  IPC0bits.INT0IP = a;
TN_BFA(TN_BFA_WR, IPC0, INT0IP, a);

//-- Read the value that is stored in the INT0IP field of the IPC0
//  register, to the int variable a:
//  int a = IPC0bits.INT0IP;
a = TN_BFA(TN_BFA_RD, IPC0, INT0IP);
```

Definition at line 154 of file `tn_arch_pic32_bfa.h`.

20.6.2.7 `#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"> • TN_BFA_WR - write bit field • TN_BFA_RD - read bit field • TN_BFA_SET - set bits by mask • TN_BFA_CLR - clear bits by mask • TN_BFA_INV - invert bits by mask
<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> != TN_BFA_RD . Meaning depends on the <i>comm</i> , see comments for specific command: TN_BFA_WR , 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 268 of file `tn_arch_pic32_bfa.h`.

20.7 arch/tn_arch.h File Reference

20.7.1 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.
- [TN_UWord * _tn_arch_stack_top_get](#) ([TN_UWord](#) *stack_low_address, int stack_size)
Should return top of the stack.
- [TN_UWord * _tn_arch_stack_bottom_empty_get](#) ([TN_UWord](#) *stack_top, int stack_size)
This function should return address of bottom empty element of stack, it is needed for software stack overflow control (see [TN_STACK_OVERFLOW_CHECK](#)).

- `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.
- `void _tn_arch_sys_start (TN_UWord *int_stack, unsigned int int_stack_size)`
Performs first context switch to the first task (`_tn_next_task_to_run` is already set to needed task).

20.7.2 Function Documentation

20.7.2.1 void tn_arch_int_dis (void)

Unconditionally disable *system interrupts*.

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

20.7.2.2 void tn_arch_int_en (void)

Unconditionally enable interrupts.

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

20.7.2.3 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\(\)](#)

20.7.2.4 void tn_arch_sr_restore (TN_UWord sr)

Restore previously saved status register.

Parameters

<code>sr</code>	status register value previously from tn_arch_sr_save_int_dis()
-----------------	---

See also

[tn_arch_sr_save_int_dis\(\)](#)

20.7.2.5 `TN_UWord* _tn_arch_stack_top_get (TN_UWord * stack_low_address, int stack_size)`

Should return top of the stack.

A stack implementation is characterized by two attributes:

- Where the stack pointer points:
 - *Full stack*: The stack pointer points to the last full location;
 - *Empty stack*: The stack pointer points to the first empty location.
- How the stack grows:
 - *Descending stack*: The stack grows downward (i.e., toward lower memory addresses);
 - *Ascending stack*: The stack grows upward (i.e., toward higher memory addresses).

So, depending on the stack implementation used in the particular architecture, the value returned from this function can be one of the following:

- $(\text{stack_low_address} - 1)$ (*Full ascending stack*)
- $(\text{stack_low_address} + \text{stack_size})$ (*Full descending stack*)
- $(\text{stack_low_address})$ (*Empty ascending stack*)
- $(\text{stack_low_address} + \text{stack_size} - 1)$ (*Empty descending stack*)

Parameters

<i>stack_low_address</i>	Start address of the stack array.
<i>stack_size</i>	Size of the stack in TN_UWord -s, not in bytes.

20.7.2.6 `TN_UWord* _tn_arch_stack_bottom_empty_get (TN_UWord * stack_top, int stack_size)`

This function should return address of bottom empty element of stack, it is needed for software stack overflow control (see [TN_STACK_OVERFLOW_CHECK](#)).

For details on various hardware stack implementations, refer to [_tn_arch_stack_top_get\(\)](#).

Depending on the stack implementation used in the particular architecture, the value returned from this function can be one of the following:

- $(\text{stack_top} - \text{stack_size})$ (*Full descending stack*)
- $(\text{stack_top} + \text{stack_size})$ (*Full ascending stack*)
- $(\text{stack_top} - \text{stack_size} + 1)$ (*Empty descending stack*)
- $(\text{stack_top} + \text{stack_size} - 1)$ (*Empty ascending stack*)

Parameters

<i>stack_top</i>	Top of the stack, returned by _tn_arch_stack_top_get() .
<i>stack_size</i>	Size of the stack in TN_UWord -s, not in bytes.

20.7.2.7 `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)

20.7.2.8 `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*.

20.7.2.9 `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*.

20.7.2.10 `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. Otherwise (if some ISR is currently running), context switch keeps pending until all ISR return.

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;

- if preprocessor macro `_TN_ON_CONTEXT_SWITCH_HANDLER` is non-zero, call `_tn_sys_on_↵context_switch(_tn_curr_run_task, _tn_next_task_to_run);`.
- 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
```

20.7.2.11 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()` (well, it is actually called indirectly but from `_tn_arch_sys_↵start()`);
- At task exit, inside `tn_task_exit()`.

This function doesn't need to 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:

- if preprocessor macro `_TN_ON_CONTEXT_SWITCH_HANDLER` is non-zero, call `_tn_sys_on_↵context_switch(_tn_curr_run_task, _tn_next_task_to_run);`.
- 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
```

20.7.2.12 void _tn_arch_sys_start (TN_UWord * int_stack, unsigned int int_stack_size)

Performs first context switch to the first task (`_tn_next_task_to_run` is already set to needed task).

Typically, this function just calls `_tn_arch_context_switch_now_nosave()`, but it also can perform any architecture-dependent actions first, if needed.

20.8 core/tn_cfg_dispatch.h File Reference

20.8.1 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))`.
- `#define _TN_ON_CONTEXT_SWITCH_HANDLER 1`
Internal kernel definition: set to non-zero if `_tn_sys_on_context_switch()` should be called on context switch.
- `#define _TN_STACK_OVERFLOW_SIZE_ADD (TN_STACK_OVERFLOW_CHECK ? 1 : 0)`
If `TN_STACK_OVERFLOW_CHECK` is set, we have 1-word overhead for each task stack.

20.8.2 Macro Definition Documentation

20.8.2.1 `#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`.

20.8.2.2 `#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`.

20.8.2.3 `#define _TN_ON_CONTEXT_SWITCH_HANDLER 1`

Internal kernel definition: set to non-zero if `_tn_sys_on_context_switch()` should be called on context switch.

Currently, the only actual handler is available: profiler (see [TN_PROFILER](#)). In the future, software stack overflow check will be implemented as well.

Definition at line 187 of file `tn_cfg_dispatch.h`.

20.9 core/tn_common.h File Reference

20.9.1 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_TickCnt`

Type for system tick count, it is used by the kernel to represent absolute tick count value as well as relative timeouts.

Enumerations

- `enum TN_ObjId {`
`TN_ID_NONE = (unsigned int)0x0, TN_ID_TASK = (unsigned int)0x47ABCF69, TN_ID_SEMAPHORE = (un-`
`signed int)0x6FA173EB, TN_ID_EVENTGRP = (unsigned int)0x5E224F25,`
`TN_ID_DATAQUEUE = (unsigned int)0x0C8A6C89, TN_ID_FSMEMORYPOOL = (unsigned int)0x26B7C`
`E8B, TN_ID_MUTEX = (unsigned int)0x17129E45, TN_ID_TIMER = (unsigned int)0x1A937FBC,`
`TN_ID_EXCHANGE = (unsigned int)0x32b7c072, TN_ID_EXCHANGE_LINK = (unsigned int)0x24d36f35 }`

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

Result code returned by kernel services.

20.9.2 Macro Definition Documentation

20.9.2.1 `#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_fmem_create()` for usage example.

Definition at line 223 of file `tn_common.h`.

20.9.2.2 `#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 247 of file `tn_common.h`.

20.9.3 Typedef Documentation

20.9.3.1 typedef unsigned long TN_TickCnt

Type for system tick count, it is used by the kernel to represent absolute tick count value as well as relative timeouts. When it is used as a timeout value, it represents the 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 the `tn_task_release_wait()` function is discouraged though)

Definition at line 188 of file `tn_common.h`.

20.9.4 Enumeration Type Documentation

20.9.4.1 enum TN_ObjId

Magic number for object validity verification.

Enumerator

TN_ID_NONE id for invalid object
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 link

Definition at line 65 of file `tn_common.h`.

20.9.4.2 enum TN_RCode

Result code returned by kernel services.

Enumerator

TN_RC_OK Successful operation.

TN_RC_TIMEOUT Timeout (consult [TN_TickCnt](#) for details).

See also

[TN_TickCnt](#)

TN_RC_OVERFLOW This code is returned in the following cases:



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

See also

[tn_sem.h](#)

[tn_fmem.h](#)

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

-  - function can be called from task;
-  - function can be called from ISR.

See also

[tn_sys_context_get\(\)](#)

enum [TN_Context](#)

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

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

See also

[TN_CHECK_PARAM](#)

TN_RC_ILLEGAL_USE Illegal usage. Returned in the following cases:

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

See also

[tn_mutex.h](#)

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

See also

[TN_CHECK_PARAM](#)

TN_RC_DELETED Object for whose event task was waiting is deleted.

TN_RC_FORCED Task was released from waiting forcibly because some other task called [tn_task_↔release_wait\(\)](#)

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

Definition at line 81 of file [tn_common.h](#).

20.10 core/tn_dqueue.h File Reference

20.10.1 Detailed Description

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

A data queue also has an associated wait queue each for sending (`wait_send` queue) and for receiving (`wait_receive` queue). A task that sends a data element tries to put the data element into the FIFO. If there is no space left in the FIFO, the task is switched to the waiting state and placed in the data queue's `wait_send` queue until space appears (another task gets a data element from the data queue).

A task that receives a data element tries to get a data element from the FIFO. If the FIFO is empty (there is no data in the data queue), the task is switched to the waiting state and placed in the data queue's `wait_receive` queue until data element arrive (another task puts some data element into the data queue). To use a data queue just for the synchronous message passing, set size of the FIFO to 0. The data element to be sent and received can be interpreted as a pointer or an integer and may have value 0 (`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.

TNeo 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_TickCnt](#) 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_TickCnt](#) 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.

20.10.2 Function Documentation

20.10.2.1 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 TN_DQueue .
<i>data_fifo</i>	pointer to already allocated array of <code>void *</code> to store data queue items. Can be TN_NULL .
<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](#).

20.10.2.2 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](#).

20.10.2.3 enum [TN_RCode tn_queue_send](#) (struct [TN_DQueue](#) * dque, void * p_data, [TN_TickCnt](#) 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_TickCnt](#).



(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 TN_TickCnt

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

See also

[TN_TickCnt](#)

20.10.2.4 `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)

20.10.2.5 `enum TN_RCode tn_queue_isend_polling (struct TN_DQueue * dque, void * p_data)`

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



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

20.10.2.6 `enum TN_RCode tn_queue_receive (struct TN_DQueue * dque, void ** pp_data, TN_TickCnt timeout)`

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

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

If there are task(s) in the data queue's `wait_send` list, first one gets removed from the head of `wait_send` list, becomes runnable and puts the data entry, stored in this task, to the tail of data FIFO. If there are no entries in the data FIFO and there are no tasks in the `wait_send` list, behavior depends on the `timeout` value: refer to [TN_TickCnt](#).



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

Parameters

<i>dque</i>	pointer to data queue to receive data from
<i>pp_data</i>	pointer to location to store the value
<i>timeout</i>	refer to TN_TickCnt

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

See also

[TN_TickCnt](#)

20.10.2.7 `enum TN_RCode tn_queue_receive_polling (struct TN_DQueue * dque, void ** pp_data)`

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



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

20.10.2.8 `enum TN_RCode tn_queue_ireceive_polling (struct TN_DQueue * dque, void ** pp_data)`

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



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

20.10.2.9 `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.

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)

20.10.2.10 `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)

20.11 core/tn_eventgrp.h File Reference

20.11.1 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.

20.11.2 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.

TNeo 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), `TN_EVENTGRP_WMODE_AUTOCLR` = (1 << 2) }
Events waiting mode that should be given to `tn_eventgrp_wait()` and friends.
- enum `TN_EGrpOp` { `TN_EVENTGRP_OP_SET`, `TN_EVENTGRP_OP_CLEAR`, `TN_EVENTGRP_OP_TOGGLE` }
Modify operation: set, clear or toggle.
- enum `TN_EGrpAttr` { `TN_EVENTGRP_ATTR_SINGLE` = (1 << 0), `TN_EVENTGRP_ATTR_MULTI` = (1 << 1), `TN_EVENTGRP_ATTR_CLR` = (1 << 2), `TN_EVENTGRP_ATTR_NONE` = (0) }
Attributes that could be given to the event group object.

Functions

- enum `TN_RCode` `tn_eventgrp_create_wattr` (struct `TN_EventGrp` *eventgrp, enum `TN_EGrpAttr` attr, `TN_UWord` initial_pattern)
The same as `tn_eventgrp_create()`, but takes additional argument: `attr`.
- static `TN_INLINE` 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_EGrpWaitMode` wait_mode, `TN_UWord` *p_flags_pattern, `TN_TickCnt` 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_EGrpWaitMode` 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_EGrpWaitMode` 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.

20.11.3 Enumeration Type Documentation

20.11.3.1 enum `TN_EGrpWaitMode`

Events waiting mode that should be given to `tn_eventgrp_wait()` and friends.

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.
- `TN_EVENTGRP_WMODE_AUTOCLR`** When a task **successfully** ends waiting for event bit(s), these bits get cleared atomically and automatically. Other bits stay unchanged.

Definition at line 124 of file `tn_eventgrp.h`.

20.11.3.2 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 144 of file `tn_eventgrp.h`.

20.11.3.3 enum TN_EGrpAttr

Attributes that could be given to the event group object.

Makes sense if only `TN_OLD_EVENT_API` option is non-zero; otherwise, there's just one dummy attribute available: `TN_EVENTGRP_ATTR_NONE`.

Enumerator

Attention

TN_EVENTGRP_ATTR_SINGLE deprecated. Available if only `TN_OLD_EVENT_API` option is non-zero.

Indicates that only one task could wait for events in this event group. This flag is mutually exclusive with `TN_EVENTGRP_ATTR_MULTI` flag.

Attention

TN_EVENTGRP_ATTR_MULTI deprecated. Available if only `TN_OLD_EVENT_API` option is non-zero.

Indicates that multiple tasks could wait for events in this event group. This flag is mutually exclusive with `TN_EVENTGRP_ATTR_SINGLE` flag.

Attention

TN_EVENTGRP_ATTR_CLR strongly deprecated. Available if only `TN_OLD_EVENT_API` option is non-zero. Use `TN_EVENTGRP_WMODE_AUTOCLR` instead.

Can be specified only in conjunction with `TN_EVENTGRP_ATTR_SINGLE` flag. Indicates that **ALL** flags in this event group should be cleared when task successfully waits for any event in it.

This actually makes little sense to clear ALL events, but this is what compatibility mode is for (see `TN_OLD_EVENT_API`)

TN_EVENTGRP_ATTR_NONE Dummy attribute that does not change anything. It is needed only for the assistance of the events compatibility mode (see `TN_OLD_EVENT_API`)

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

20.11.4 Function Documentation

20.11.4.1 enum TN_RCode tn_eventgrp_create_wattr (struct TN_EventGrp * eventgrp, enum TN_EGrpAttr attr, TN_UWord initial_pattern)

The same as `tn_eventgrp_create()`, but takes additional argument: `attr`.

It makes sense if only `TN_OLD_EVENT_API` option is non-zero.

Parameters

<i>eventgrp</i>	Pointer to already allocated struct TN_EventGrp
<i>attr</i>	Attributes for that particular event group object, see struct TN_EGrpAttr
<i>initial_pattern</i>	Initial events pattern.

20.11.4.2 **static _TN_INLINE** enum **TN_RCode** **tn_eventgrp_create** (struct **TN_EventGrp** * *eventgrp*, **TN_UWord** *initial_pattern*) [static]

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 TN_EventGrp
<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](#).

Definition at line 304 of file [tn_eventgrp.h](#).

20.11.4.3 **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](#).

20.11.4.4 **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_TickCnt** *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_TickCnt](#).



(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 all the event bits from <i>wait_pattern</i> to be set, or for just any of them (see enum TN_EGrpWaitMode)
<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 TN_TickCnt

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

20.11.4.5 `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)

20.11.4.6 `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)

20.11.4.7 `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 TN_EGrpOp
<i>pattern</i>	Events pattern to be applied (depending on <i>operation</i> 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](#).

20.11.4.8 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)

20.12 core/tn_fmem.h File Reference

20.12.1 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_fmem.h](#).

Data Structures

- struct [TN_FMem](#)
Fixed memory blocks pool.
- struct [TN_FMemTaskWait](#)
FMem-specific fields related to waiting task, to be included in struct [TN_Task](#).

Macros

- #define [TN_FMEM_BUF_DEF](#)(name, item_type, size)
Convenience macro for the definition of buffer for memory pool.

Functions

- enum [TN_RCode](#) [tn_fmem_create](#) (struct [TN_FMem](#) *`fmem`, void *`start_addr`, unsigned int `block_size`, int `blocks_cnt`)
Construct fixed memory blocks pool.
- enum [TN_RCode](#) [tn_fmem_delete](#) (struct [TN_FMem](#) *`fmem`)
Destruct fixed memory blocks pool.
- enum [TN_RCode](#) [tn_fmem_get](#) (struct [TN_FMem](#) *`fmem`, void **`p_data`, [TN_TickCnt](#) `timeout`)
Get memory block from the pool.
- enum [TN_RCode](#) [tn_fmem_get_polling](#) (struct [TN_FMem](#) *`fmem`, void **`p_data`)
The same as [tn_fmem_get\(\)](#) with zero timeout.
- enum [TN_RCode](#) [tn_fmem_iget_polling](#) (struct [TN_FMem](#) *`fmem`, void **`p_data`)
The same as [tn_fmem_get\(\)](#) with zero timeout, but for using in the ISR.
- enum [TN_RCode](#) [tn_fmem_release](#) (struct [TN_FMem](#) *`fmem`, void *`p_data`)
Release memory block back to the pool.
- enum [TN_RCode](#) [tn_fmem_irelease](#) (struct [TN_FMem](#) *`fmem`, void *`p_data`)
The same as [tn_fmem_get\(\)](#), but for using in the ISR.

20.12.2 Macro Definition Documentation

20.12.2.1 #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 tn_fmем_create() 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`.

20.12.3 Function Documentation

20.12.3.1 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_fmp` field should not contain [TN_ID_FSMEMORYPOOL](#), otherwise, [TN_RC_WPARAM](#) is returned.

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

For the definition of buffer, convenience macro [TN_FMEM_BUF_DEF\(\)](#) was invented.

Typical definition looks as follows:

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

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

//-- define buffer for memory pool
TN_FMEM_BUF_DEF(my_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 <code>struct TN_FMem</code> .
<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 <code>sizeof(TN_UWord)</code> , 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](#)

20.12.3.2 `enum TN_RCode tn_fmem_delete (struct TN_FMem * fmem)`

Destruct fixed memory blocks pool.

All tasks that wait for free memory block become runnable with [TN_RC_DELETED](#) code returned.



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

Parameters

<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](#).

20.12.3.3 `enum TN_RCode tn_fmem_get (struct TN_FMem * fmem, void ** p_data, TN_TickCnt 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_TickCnt](#).



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

Parameters

<i>fmem</i>	Pointer to memory pool
<i>p_data</i>	Address of the <code>(void *)</code> to which received block address will be saved
<i>timeout</i>	Refer to TN_TickCnt

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

20.12.3.4 `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)

20.12.3.5 `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)

20.12.3.6 `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`.

20.12.3.7 `enum TN_RCode tn_fmem_irelease (struct TN_FMem * fmem, void * p_data)`

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



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

20.13 core/tn_list.h File Reference

20.13.1 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.

20.14 core/tn_mutex.h File Reference

20.14.1 Detailed Description

A mutex is an object used to protect shared resources.

There is a lot of confusion about the differences between semaphores and mutexes, so, it's highly recommended that you read a small article by Michael Barr: [Mutexes and Semaphores Demystified](#).

Very short:

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

A semaphore, on the other hand, is a *signaling mechanism*. It's quite legal and encouraged for a semaphore to be acquired in task A, and then signaled from task B or even from an 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 TNeo:

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

The priority inheritance protocol solves the priority inversion problem, but doesn't prevent deadlocks. However, 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_TickCnt](#) 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.

20.14.2 Enumeration Type Documentation

20.14.2.1 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](#).

20.14.3 Function Documentation

20.14.3.1 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 struct TN_Mutex
<i>protocol</i>	Mutex protocol: priority ceiling or priority inheritance. See enum TN_MutexProtocol .
<i>ceil_priority</i>	Used if only <code>protocol</code> is TN_MUTEX_PROT_CEILING : maximum priority of the task that may lock the mutex.

Returns

- [TN_RC_OK](#) if mutex was successfully created;
- If [TN_CHECK_PARAM](#) is non-zero, additional return code is available: [TN_RC_WPARAM](#).

20.14.3.2 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](#).

20.14.3.3 enum [TN_RCode](#) [tn_mutex_lock](#) (struct [TN_Mutex](#) * *mutex*, [TN_TickCnt](#) *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_TickCnt](#).



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

Parameters

<i>mutex</i>	mutex to lock
<i>timeout</i>	refer to TN_TickCnt

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

See also

[TN_MutexProtocol](#)

20.14.3.4 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)

20.14.3.5 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](#).

20.15 core/tn_oldsymbols.h File Reference

20.15.1 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 _TN_EVENT TN_EventGrp](#)
old TNKernel struct name of [TN_EventGrp](#), available if only [TN_OLD_EVENT_API](#) is non-zero
- [#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_fmem_get_ipolling` `tn_fmem_iget_polling`
 - old TNKernel name of `tn_fmem_iget_polling`*
- #define `tn_queue_ireceive` `tn_queue_ireceive_polling`
 - old TNKernel name of `tn_queue_ireceive_polling`*
- #define `tn_start_system` `tn_sys_start`
 - old TNKernel name of `tn_sys_start`*
- #define `tn_sys_tslice_ticks` `tn_sys_tslice_set`
 - old TNKernel name of `tn_sys_tslice_set`*
- #define `align_attr_start` `TN_ARCH_STK_ATTR_BEFORE`
 - old TNKernel name of `TN_ARCH_STK_ATTR_BEFORE`*
- #define `align_attr_end` `TN_ARCH_STK_ATTR_AFTER`
 - old TNKernel name of `TN_ARCH_STK_ATTR_AFTER`*
- #define `tn_cpu_int_disable` `tn_arch_int_dis`
 - old TNKernel name of `tn_arch_int_dis`*
- #define `tn_cpu_int_enable` `tn_arch_int_en`
 - old TNKernel name of `tn_arch_int_en`*
- #define `tn_cpu_save_sr` `tn_arch_sr_save_int_dis`
 - old TNKernel name of `tn_arch_sr_save_int_dis`*
- #define `tn_cpu_restore_sr` `tn_arch_sr_restore`
 - old TNKernel name of `tn_arch_sr_restore`*
- #define `tn_disable_interrupt` `TN_INT_DIS_SAVE`
 - old TNKernel name of `TN_INT_DIS_SAVE`*
- #define `tn_enable_interrupt` `TN_INT_RESTORE`
 - old TNKernel name of `TN_INT_RESTORE`*
- #define `tn_idisable_interrupt` `TN_INT_IDIS_SAVE`
 - old TNKernel name of `TN_INT_IDIS_SAVE`*
- #define `tn_ienable_interrupt` `TN_INT_IRESTORE`
 - old TNKernel name of `TN_INT_IRESTORE`*
- #define `tn_chk_irq_disabled` `TN_IS_INT_DISABLED`
 - old TNKernel name of `TN_IS_INT_DISABLED`*
- #define `TN_NUM_PRIORITY` `TN_PRIORITIES_CNT`
 - old TNKernel name of `TN_PRIORITIES_CNT`*

- `#define _TN_BITS_IN_INT TN_INT_WIDTH`
old TNKernel name of `TN_INT_WIDTH`
- `#define TN_ALIG sizeof(TN_UWord)`
old TNKernel name for `sizeof(TN_UWord)`
- `#define NO_TIME_SLICE TN_NO_TIME_SLICE`
old TNKernel name for `TN_NO_TIME_SLICE`
- `#define MAX_TIME_SLICE TN_MAX_TIME_SLICE`
old TNKernel name for `TN_MAX_TIME_SLICE`
- `#define TN_TASK_STACK_DEF TN_STACK_ARR_DEF`
old name for `TN_STACK_ARR_DEF`
- `#define TN_Timeout TN_TickCnt`
old name for `TN_TickCnt`
- `#define TN_EVENT_ATTR_SINGLE TN_EVENTGRP_ATTR_SINGLE`
- `#define TN_EVENT_ATTR_MULTI TN_EVENTGRP_ATTR_MULTI`
- `#define TN_EVENT_ATTR_CLR TN_EVENTGRP_ATTR_CLR`
- `#define tn_event_create(ev, attr, pattern) tn_eventgrp_create_wattr((ev), (enum TN_EGrpAttr)(attr), (pattern))`
- `#define tn_event_delete tn_eventgrp_delete`
- `#define tn_event_wait tn_eventgrp_wait`
- `#define tn_event_wait_polling tn_eventgrp_wait_polling`
- `#define tn_event_await tn_eventgrp_await_polling`
- `#define tn_event_set(ev, pattern) tn_eventgrp_modify((ev), TN_EVENTGRP_OP_SET, (pattern))`
- `#define tn_event_iset(ev, pattern) tn_eventgrp_imodify((ev), TN_EVENTGRP_OP_SET, (pattern))`
- `#define tn_event_clear(ev, pattern) tn_eventgrp_modify((ev), TN_EVENTGRP_OP_CLEAR, (~(pattern)))`
- `#define tn_event_iclear(ev, pattern) tn_eventgrp_imodify((ev), TN_EVENTGRP_OP_CLEAR, (~(pattern)))`

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`
- `typedef struct TN_EventGrp TN_EVENT`
old TNKernel name of `TN_EventGrp` available if only `TN_OLD_EVENT_API` is non-zero

20.15.2 Macro Definition Documentation

20.15.2.1 `#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 144 of file `tn_oldsymbols.h`.

20.15.2.2 #define TN_EVENT_ATTR_SINGLE TN_EVENTGRP_ATTR_SINGLE**Attention**

Deprecated. Available if only [TN_OLD_EVENT_API](#) option is non-zero.

Old name for [TN_EVENTGRP_ATTR_SINGLE](#),

Definition at line 356 of file `tn_oldsymbols.h`.

20.15.2.3 #define TN_EVENT_ATTR_MULTI TN_EVENTGRP_ATTR_MULTI**Attention**

Deprecated. Available if only [TN_OLD_EVENT_API](#) option is non-zero.

Old name for [TN_EVENTGRP_ATTR_MULTI](#),

Definition at line 362 of file `tn_oldsymbols.h`.

20.15.2.4 #define TN_EVENT_ATTR_CLR TN_EVENTGRP_ATTR_CLR**Attention**

Deprecated. Available if only [TN_OLD_EVENT_API](#) option is non-zero.

Old name for [TN_EVENTGRP_ATTR_CLR](#),

Definition at line 368 of file `tn_oldsymbols.h`.

20.15.2.5 #define tn_event_create(ev, attr, pattern) tn_eventgrp_create_wattr((ev), (enum TN_EGrpAttr)(attr), (pattern))**Attention**

Deprecated. Available if only [TN_OLD_EVENT_API](#) option is non-zero.

Old name for [tn_eventgrp_create_wattr\(\)](#),

Definition at line 374 of file `tn_oldsymbols.h`.

20.15.2.6 #define tn_event_delete tn_eventgrp_delete**Attention**

Deprecated. Available if only [TN_OLD_EVENT_API](#) option is non-zero.

Old name for [tn_eventgrp_delete\(\)](#),

Definition at line 381 of file `tn_oldsymbols.h`.

20.15.2.7 #define tn_event_wait tn_eventgrp_wait**Attention**

Deprecated. Available if only [TN_OLD_EVENT_API](#) option is non-zero.

Old name for [tn_eventgrp_wait\(\)](#),

Definition at line 387 of file `tn_oldsymbols.h`.

20.15.2.8 #define tn_event_wait_polling tn_eventgrp_wait_polling

Attention

Deprecated. Available if only `TN_OLD_EVENT_API` option is non-zero.

Old name for `tn_eventgrp_wait_polling()`,

Definition at line 393 of file `tn_oldsymbols.h`.

20.15.2.9 #define tn_event_await tn_eventgrp_await_polling

Attention

Deprecated. Available if only `TN_OLD_EVENT_API` option is non-zero.

Old name for `tn_eventgrp_await_polling()`,

Definition at line 399 of file `tn_oldsymbols.h`.

20.15.2.10 #define tn_event_set(ev, pattern) tn_eventgrp_modify ((ev), TN_EVENTGRP_OP_SET, (pattern))

Attention

Deprecated. Available if only `TN_OLD_EVENT_API` option is non-zero.

Old TNKernel-compatible way of calling `tn_eventgrp_modify` (event, `TN_EVENTGRP_OP_SET`, pattern)

Definition at line 406 of file `tn_oldsymbols.h`.

20.15.2.11 #define tn_event_iset(ev, pattern) tn_eventgrp_imodify((ev), TN_EVENTGRP_OP_SET, (pattern))

Attention

Deprecated. Available if only `TN_OLD_EVENT_API` option is non-zero.

Old TNKernel-compatible way of calling `tn_eventgrp_imodify` (event, `TN_EVENTGRP_OP_SET`, pattern)

Definition at line 413 of file `tn_oldsymbols.h`.

20.15.2.12 #define tn_event_clear(ev, pattern) tn_eventgrp_modify ((ev), TN_EVENTGRP_OP_CLEAR, (~(pattern)))

Attention

Deprecated. Available if only `TN_OLD_EVENT_API` option is non-zero.

Old TNKernel-compatible way of calling `tn_eventgrp_modify` (event, `TN_EVENTGRP_OP_CLEAR`, (~pattern))

Attention

Unlike `tn_eventgrp_modify()`, the pattern should be inverted!

Definition at line 422 of file `tn_oldsymbols.h`.

```
20.15.2.13 #define tn_event_iclear( ev, pattern ) tn_eventgrp_imodify((ev), TN_EVENTGRP_OP_CLEAR,
    (~(pattern)))
```

Attention

Deprecated. Available if only `TN_OLD_EVENT_API` option is non-zero.

Old TNKernel-compatible way of calling `tn_eventgrp_imodify (event, TN_EVENTGRP_OP_CLEAR, (~pattern))`

Attention

Unlike `tn_eventgrp_modify()`, the pattern should be inverted!

Definition at line 431 of file `tn_oldsymbols.h`.

20.16 core/tn_sem.h File Reference

20.16.1 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_TickCnt` 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.

20.16.2 Function Documentation

20.16.2.1 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 TN_Sem
<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](#).

20.16.2.2 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](#).

20.16.2.3 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](#).

20.16.2.4 enum [TN_RCode](#) [tn_sem_signal](#) (struct [TN_Sem](#) * *sem*)

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



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

20.16.2.5 enum [TN_RCode](#) [tn_sem_wait](#) (struct [TN_Sem](#) * *sem*, [TN_TickCnt](#) *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_TickCnt](#).



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

Parameters

<i>sem</i>	semaphore to wait for
<i>timeout</i>	refer to TN_TickCnt

Returns

- [TN_RC_OK](#) if waiting was successfull
- Other possible return codes depend on `timeout` value, refer to [TN_TickCnt](#)
- If [TN_CHECK_PARAM](#) is non-zero, additional return codes are available: [TN_RC_WPARAM](#) and [TN_RC_INVALID_OBJ](#).

20.16.2.6 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)

20.16.2.7 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)

20.17 core/tn_sys.h File Reference

20.17.1 Detailed Description

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

Definition in file [tn_sys.h](#).

Data Structures

- struct [_TN_BuildCfg](#)
Structure with build-time configurations values; it is needed for run-time check which ensures that build-time options for the kernel match ones for the application.

Macros

- #define [TN_STACK_ARR_DEF](#)(name, size)
Convenience macro for the definition of stack array.
- #define [_TN_BUILD_CFG_ARCH_STRUCT_FILL](#)(_p_struct)
For internal kernel usage: helper macro that fills architecture-dependent values.
- #define [_TN_BUILD_CFG_STRUCT_FILL](#)(_p_struct)
For internal kernel usage: fill the structure [_TN_BuildCfg](#) with current build-time configuration values.
- #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_CBStackOverflow](#))(struct [TN_Task](#) *task)
User-provided callback function that is called when the kernel detects stack overflow (see [TN_STACK_OVERFLOW_CHECK](#)).
- 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_CBUUserTaskCreate](#) *cb_user_task_create, [TN_CBIdle](#) *cb_idle)
Initial TNeo 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](#)).
- [TN_TickCnt](#) [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)
- void [tn_callback_stack_overflow_set](#) ([TN_CBStackOverflow](#) *cb)
Set callback function that is called when the kernel detects stack overflow (see [TN_STACK_OVERFLOW_CHECK](#)).
- 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_INLINE](#) [TN_BOOL](#) [tn_is_task_context](#) (void)
Returns whether current system context is [TN_CONTEXT_TASK](#)
- static [_TN_INLINE](#) [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.
- void [tn_callback_dyn_tick_set](#) ([TN_CBTickSchedule](#) *cb_tick_schedule, [TN_CBTickCntGet](#) *cb_tick_cnt_get)
Available if only [TN_DYNAMIC_TICK](#) is set.

20.17.2 Macro Definition Documentation

20.17.2.1 #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 (TN_UWord), not in bytes.

Definition at line 87 of file [tn_sys.h](#).

20.17.2.2 #define _TN_BUILD_CFG_ARCH_STRUCT_FILL(_p_struct)

For internal kernel usage: helper macro that fills architecture-dependent values.

This macro is used by `_TN_BUILD_CFG_STRUCT_FILL()` only.

Definition at line 107 of file `tn_sys.h`.

20.17.2.3 #define _TN_BUILD_CFG_STRUCT_FILL(_p_struct)

Value:

```
{
    memset((p_struct), 0x00, sizeof(*(_p_struct)));

    (_p_struct)->priorities_cnt      = TN_PRIORITIES_CNT;
    (_p_struct)->check_param        = TN_CHECK_PARAM;
    (_p_struct)->debug              = TN_DEBUG;
    (_p_struct)->use_mutexes        = TN_USE_MUTEXES;
    (_p_struct)->mutex_rec          = TN_MUTEX_REC;
    (_p_struct)->mutex_deadlock_detect = TN_MUTEX_DEADLOCK_DETECT;
    (_p_struct)->tick_lists_cnt_minus_one = (TN_TICK_LISTS_CNT - 1);
    (_p_struct)->api_make_alig_arg  = TN_API_MAKE_ALIG_ARG;
    (_p_struct)->profiler           = TN_PROFILER;
    (_p_struct)->profiler_wait_time = TN_PROFILER_WAIT_TIME;
    (_p_struct)->stack_overflow_check = TN_STACK_OVERFLOW_CHECK;
    (_p_struct)->dynamic_tick       = TN_DYNAMIC_TICK;
    (_p_struct)->old_events_api     = TN_OLD_EVENT_API;

    _TN_BUILD_CFG_ARCH_STRUCT_FILL(_p_struct);
}
```

For internal kernel usage: fill the structure `_TN_BuildCfg` with current build-time configuration values.

Parameters

<code>_p_struct</code>	Pointer to struct <code>_TN_BuildCfg</code>
------------------------	---

Definition at line 119 of file `tn_sys.h`.

20.17.3 Typedef Documentation

20.17.3.1 typedef void(TN_CBUUserTaskCreate)(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_CBUUserTaskCreate()` 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 283 of file `tn_sys.h`.

20.17.3.2 `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 298 of file `tn_sys.h`.

20.17.3.3 `typedef void(TN_CBStackOverflow)(struct TN_Task *task)`

User-provided callback function that is called when the kernel detects stack overflow (see [TN_STACK_OVERFLOW_CHECK](#)).

Parameters

<i>task</i>	Task whose stack is overflowed
-------------	--------------------------------

Definition at line 307 of file `tn_sys.h`.

20.17.3.4 `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->deadlock_list</code> .
<i>task</i>	task that is involved in deadlock. You may find out other tasks involved by means of <code>task->deadlock_list</code> .

Definition at line 327 of file `tn_sys.h`.

20.17.4 Enumeration Type Documentation

20.17.4.1 `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 221 of file tn_sys.h.

20.17.4.2 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 237 of file tn_sys.h.

20.17.5 Function Documentation

20.17.5.1 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 TNeo 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

<i>idle_task_stack</i>	Pointer to array for idle task stack. User must either use the macro TN_STACK_ARR_DEFINE() for the definition of stack array, or allocate it manually as an array of TN_UWord with TN_ARCH_STK_ATTR_BEFORE and TN_ARCH_STK_ATTR_AFTER macros.
<i>idle_task_stack_size</i>	Size of idle task stack, in words (TN_UWord)
<i>int_stack</i>	Pointer to array for interrupt stack. User must either use the macro TN_STACK_ARR_DEFINE() for the definition of stack array, or allocate it manually as an array of TN_UWord with TN_ARCH_STK_ATTR_BEFORE and TN_ARCH_STK_ATTR_AFTER macros.
<i>int_stack_size</i>	Size of interrupt stack, in words (TN_UWord)
<i>cb_user_task_create</i>	Callback function that should create initial user's task, see TN_CBUserTaskCreate for details.

<i>cb_idle</i>	Callback function repeatedly called from idle task, see TN_CBIdle for details.
----------------	--

20.17.5.2 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.

20.17.5.3 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 TN_NO_TIME_SLICE for no round-robin scheduling for given priority (it's default value). Value can't be higher than TN_MAX_TIME_SLICE .

Returns

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

20.17.5.4 `TN_TickCnt` `tn_sys_time_get (void)`

Get current system ticks count.



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

Returns

Current system ticks count.

20.17.5.5 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

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

See also

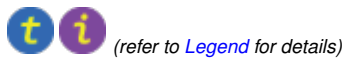
[TN_MUTEX_DEADLOCK_DETECT](#)
[TN_CBDeadlock](#) for callback function prototype

20.17.5.6 `void tn_callback_stack_overflow_set (TN_CBStackOverflow * cb)`

Set callback function that is called when the kernel detects stack overflow (see [TN_STACK_OVERFLOW_CHECK](#)).
 For function prototype, refer to [TN_CBStackOverflow](#).

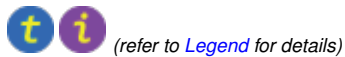
20.17.5.7 `enum TN_StateFlag tn_sys_state_flags_get (void)`

Returns current system state flags.



20.17.5.8 `enum TN_Context tn_sys_context_get (void)`

Returns system context: task or ISR.

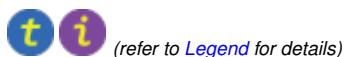


See also

enum [TN_Context](#)

20.17.5.9 `static _TN_INLINE TN_BOOL tn_is_task_context (void) [static]`

Returns whether current system context is [TN_CONTEXT_TASK](#)



Returns

`TN_TRUE` if current system context is [TN_CONTEXT_TASK](#), `TN_FALSE` otherwise.

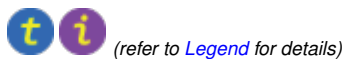
See also

[tn_sys_context_get\(\)](#)
 enum [TN_Context](#)

Definition at line 514 of file `tn_sys.h`.

20.17.5.10 `static _TN_INLINE TN_BOOL tn_is_isr_context (void) [static]`

Returns whether current system context is [TN_CONTEXT_ISR](#)



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 533 of file [tn_sys.h](#).

20.17.5.11 struct TN_Task* tn_cur_task_get (void)

Returns pointer to the currently running task.



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

20.17.5.12 TN_TaskBody* tn_cur_task_body_get (void)

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



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

20.17.5.13 void tn_callback_dyn_tick_set (TN_CBTickSchedule * cb_tick_schedule, TN_CBTickCntGet * cb_tick_cnt_get)

Available if only [TN_DYNAMIC_TICK](#) is **set**.

Set callbacks related to dynamic tick.

Attention

This function should be called **before** [tn_sys_start\(\)](#), otherwise, you'll run into run-time error [_TN_FATAL_ERROR\(\)](#).

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

Parameters

cb_tick_schedule	Pointer to callback function to schedule next time to call tn_tick_int_processing() , see TN_CBTickSchedule for the prototype.
cb_tick_cnt_get	Pointer to callback function to get current system tick counter value, see TN_CBTickCntGet for the prototype.

20.18 core/tn_tasks.h File Reference**20.18.1 Detailed Description****20.18.2 Task**

In TNeo, 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 TNeo 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.

20.18.3 Task states

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

20.18.4 Creating/starting tasks

Create task and start 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.

20.18.5 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](#).

20.18.6 Scheduling rules

TNeo 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.

20.18.7 Idle task

TNeo 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_TaskTiming](#)

Timing structure that is managed by profiler and can be read by [tn_task_profiler_timing_get\(\)](#) function.

- struct `_TN_TaskProfiler`
Internal kernel structure for profiling data of task.
- 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_WAITSUSP` = (`TN_TASK_STATE_WAIT` | `TN_TASK_STATE_SUSPEND`), `TN_TASK_STATE_DORMANT` = (1 << 3) }
Task state.
- enum `TN_WaitReason` {
`TN_WAIT_REASON_NONE`, `TN_WAIT_REASON_SLEEP`, `TN_WAIT_REASON_SEM`, `TN_WAIT_REASON_SEM_ON_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`, `TN_WAIT_REASONS_CNT` }
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_create_wname` (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, const char *name)
The same as `tn_task_create()` but with additional argument `name`, which could be very useful for debug.
- 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_TickCnt` 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_profiler_timing_get](#) (const struct [TN_Task](#) *task, struct [TN_TaskTiming](#) *tgt)
Read profiler timing data of the task.
- enum [TN_RCode tn_task_change_priority](#) (struct [TN_Task](#) *task, int new_priority)
Set new priority for task.

20.18.8 Enumeration Type Documentation

20.18.8.1 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](#).

20.18.8.2 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](#)

TN_WAIT_REASONS_CNT Wait reasons count.

Definition at line 173 of file [tn_tasks.h](#).

20.18.8.3 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 221 of file [tn_tasks.h](#).

20.18.8.4 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 236 of file [tn_tasks.h](#).

20.18.9 Function Documentation

20.18.9.1 `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

<i>task</i>	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.
<i>task_func</i>	Pointer to task body function.
<i>priority</i>	Priority for new task. NOTE: the lower value, the higher priority. Must be > 0 and $< (TN_PRIORITIES_CNT - 1)$.
<i>task_stack_low_addr</i>	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.
<i>task_stack_size</i>	Size of task stack array, in words (<code>TN_UWord</code>), not in bytes.
<i>param</i>	Parameter that is passed to <code>task_func</code> .
<i>opts</i>	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_task_create_wname()
TN_ARCH_STK_ATTR_BEFORE
TN_ARCH_STK_ATTR_AFTER
```

20.18.9.2 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

<i>task</i>	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
```

20.18.9.3 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

<i>task</i>	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
```

20.18.9.4 enum `TN_RCode` `tn_task_sleep (TN_TickCnt 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_TickCnt</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_TickCnt](#)

20.18.9.5 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`.

20.18.9.6 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)

20.18.9.7 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

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

20.18.9.8 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)

20.18.9.9 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

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

20.18.9.10 `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)

20.18.9.11 `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\(\)](#)

20.18.9.12 `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

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

20.18.9.13 `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](#).

20.18.9.14 `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

20.18.9.15 `enum TN_RCode tn_task_profiler_timing_get (const struct TN_Task * task, struct TN_TaskTiming * tgt)`

Read profiler timing data of the task.

See `struct TN_TaskTiming` for details on timing data.



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

Parameters

<i>task</i>	Task to get timing data of
<i>tgt</i>	Target structure to fill with data, should be allocated by caller

20.18.9.16 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

20.19 core/tn_timer.h File Reference

20.19.1 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 TNeo, 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.

TNeo offers two implementations of timers: static and dynamic. Refer to the page [Time ticks](#) for details.

20.19.2 Implementation of static 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, TNeo'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 TNeo 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.
- typedef void([TN_CBTickSchedule](#))(TN_TickCnt timeout)
*Available if only [TN_DYNAMIC_TICK](#) is **set**.*
- typedef TN_TickCnt([TN_CBTickCntGet](#))(void)
*Available if only [TN_DYNAMIC_TICK](#) is **set**.*

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

20.19.3 Typedef Documentation

20.19.3.1 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 <code>tn_timer_create()</code> .

Definition at line 200 of file `tn_timer.h`.

20.19.3.2 `typedef void(TN_CBTickSchedule)(TN_TickCnt timeout)`

Available if only `TN_DYNAMIC_TICK` is **set**.

Prototype of callback function that should schedule next time to call `tn_tick_int_processing()`.

See `tn_callback_dyn_tick_set()`

Parameters

<i>timeout</i>	<p>Timeout after which <code>tn_tick_int_processing()</code> should be called next time. Note the following:</p> <ul style="list-style-type: none"> • It might be <code>TN_WAIT_INFINITE</code>, which means that there are no active timeouts, and so, there's no need for tick interrupt at all. • It might be 0; in this case, it's <i>already</i> time to call <code>tn_tick_int_processing()</code>. You might want to set interrupt request bit then, in order to get to it as soon as possible. • In other cases, the function should schedule next call to <code>tn_tick_int_processing()</code> in the <code>timeout</code> tick periods.
----------------	---

Definition at line 269 of file `tn_timer.h`.

20.19.3.3 `typedef TN_TickCnt(TN_CBTickCntGet)(void)`

Available if only `TN_DYNAMIC_TICK` is **set**.

Prototype of callback function that should return current system tick counter value.

See `tn_callback_dyn_tick_set()`

Returns

current system tick counter value.

Definition at line 281 of file `tn_timer.h`.

20.19.4 Function Documentation

20.19.4.1 `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.

20.19.4.2 `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`.

20.19.4.3 `enum TN_RCode tn_timer_start (struct TN_Timer * timer, TN_TickCnt 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). Note 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`.

20.19.4.4 `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](#).

20.19.4.5 `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 TN_NULL . See TN_TimerFunc()
<i>p_user_data</i>	User data pointer that is given to user-provided <i>func</i> .

Returns

- [TN_RC_OK](#) if operation was successful;
- [TN_RC_WPARAM](#) if wrong params were given.

20.19.4.6 `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 TN_BOOL variable in which resulting value should be stored

Returns

- [TN_RC_OK](#) if operation was successful;
- [TN_RC_WPARAM](#) if wrong params were given.

20.19.4.7 `enum TN_RCode tn_timer_time_left (struct TN_Timer * timer, TN_TickCnt * 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

<i>timer</i>	Pointer to timer
<i>p_time_left</i>	Pointer to TN_TickCnt variable in which resulting value should be stored

Returns

- [TN_RC_OK](#) if operation was successful;
- [TN_RC_WPARAM](#) if wrong params were given.

20.20 tn.h File Reference

20.20.1 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](#).

20.21 tn_app_check.c File Reference

20.21.1 Detailed Description

If [TN_CHECK_BUILD_CFG](#) option is non-zero, this file needs to be included in the application project.

For details, see the aforementioned option [TN_CHECK_BUILD_CFG](#).

Definition in file [tn_app_check.c](#).

Functions

- void [you_should_add_file__tn_app_check_c__to_the_project](#) (void)
Dummy function that helps user to understand that he/she forgot to add file [tn_app_check.c](#) to the project.
- const struct [_TN_BuildCfg](#) * [tn_app_build_cfg_get](#) (void)
Return build configuration used for application.

20.21.2 Function Documentation

20.21.2.1 void you_should_add_file__tn_app_check_c__to_the_project (void)

Dummy function that helps user to understand that he/she forgot to add file [tn_app_check.c](#) to the project.

It is called from [tn_sys.c](#) .

Definition at line 91 of file [tn_app_check.c](#).

20.22 tn_cfg_default.h File Reference

20.22.1 Detailed Description

TNeo 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 TNeo source directory, like this:

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

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

Definition in file `tn_cfg_default.h`.

Macros

- `#define TN_CHECK_BUILD_CFG 1`
This option enables run-time check which ensures that build-time options for the kernel match ones for the application.
- `#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 TNeo 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`
*Takes effect if only `TN_DYNAMIC_TICK` is **not set**.*
- `#define TN_API_MAKE_ALIG_ARG TN_API_MAKE_ALIG_ARG_SIZE`
API option for `MAKE_ALIG()` macro.
- `#define TN_PROFILER 0`
Whether profiler functionality should be enabled.
- `#define TN_PROFILER_WAIT_TIME 0`
Whether profiler should store wait time for each wait reason.
- `#define TN_STACK_OVERFLOW_CHECK 1`
Whether software stack overflow check is enabled.
- `#define TN_DYNAMIC_TICK 0`
*Whether the kernel should use *Dynamic tick* scheme instead of *Static tick*.*
- `#define TN_OLD_EVENT_API 0`
Whether the old TNKernel events API compatibility mode is active.
- `#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:

20.22.2 Macro Definition Documentation

20.22.2.1 #define TN_CHECK_BUILD_CFG 1

This option enables run-time check which ensures that build-time options for the kernel match ones for the application.

Without this check, it is possible that you change your `tn_cfg.h` file, and just rebuild your application without rebuilding the kernel. Then, application would assume that kernel behaves accordingly to `tn_cfg.h` which was included in the application, but this is actually not true: you need to rebuild the kernel for changes to take effect.

With this option turned on, if build-time configurations don't match, you will get run-time error (`_TN_FATAL_ERROR()`) inside `tn_sys_start()`, which is much more informative than weird bugs caused by configuration mismatch.

Note: turning this option on makes sense if only you use TNeo as a separate library. If you build TNeo together with the application, both the kernel and the application always use the same `tn_cfg.h` file, therefore this option is useless.

Attention

If this option is on, your application must include the file `tn_app_check.c`.

Definition at line 107 of file `tn_cfg_default.h`.

20.22.2.2 #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 130 of file `tn_cfg_default.h`.

20.22.2.3 #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 147 of file `tn_cfg_default.h`.

20.22.2.4 `#define TN_DEBUG 0`

Allows additional internal self-checking, useful to catch internal TNeo 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 157 of file `tn_cfg_default.h`.

20.22.2.5 `#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 TNeo from scratch, it's better to avoid old names.

Definition at line 167 of file `tn_cfg_default.h`.

20.22.2.6 `#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()`
see `TN_CBDeadlock`

Definition at line 192 of file `tn_cfg_default.h`.

20.22.2.7 `#define TN_TICK_LISTS_CNT 8`

*Takes effect if only `TN_DYNAMIC_TICK` is **not set**.*

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 static 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 213 of file `tn_cfg_default.h`.

20.22.2.8 `#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_ALIG(↵ 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 245 of file `tn_cfg_default.h`.

20.22.2.9 `#define TN_PROFILER 0`

Whether profiler functionality should be enabled.

Enabling this option adds overhead to context switching and increases the size of `TN_Task` structure by about 20 bytes.

See also

```
TN_PROFILER_WAIT_TIME
tn_task_profiler_timing_get()
struct TN_TaskTiming
```

Definition at line 259 of file `tn_cfg_default.h`.

20.22.2.10 `#define TN_PROFILER_WAIT_TIME 0`

Whether profiler should store wait time for each wait reason.

Enabling this option bumps the size of `TN_Task` structure by more than 100 bytes, see `struct TN_Task↵Timing`.

Relevant if only `TN_PROFILER` is non-zero.

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

20.22.2.11 `#define TN_STACK_OVERFLOW_CHECK 1`

Whether software stack overflow check is enabled.

Enabling this option adds small overhead to context switching and system tick processing (`tn_tick_int↵processing()`). When stack overflow happens, the kernel calls user-provided callback (see `tn_callback↵_stack_overflow_set()`); if this callback is undefined, the kernel calls `_TN_FATAL_ERROR()`.

This option is on by default for all architectures except PIC24/dsPIC, since this architecture has hardware stack pointer limit.

Definition at line 295 of file `tn_cfg_default.h`.

20.22.2.12 `#define TN_OLD_EVENT_API 0`

Whether the old TNKernel events API compatibility mode is active.

Warning

Use it if only you're porting your existing TNKernel project on TNeo. Otherwise, usage of this option is strongly discouraged.

Actually, events are the most incompatible thing between TNeo and TNKernel (for some details, refer to the section [Events API is changed almost completely](#))

This option is quite useful when you're porting your existing TNKernel app to TNeo. When it is non-zero, old events symbols are available and behave just like they do in TNKernel.

The full list of what becomes available:

- Event group attributes:
 - `TN_EVENT_ATTR_SINGLE`
 - `TN_EVENT_ATTR_MULTII`
 - `TN_EVENT_ATTR_CLR`
- Functions:
 - `tn_event_create()`
 - `tn_event_delete()`
 - `tn_event_wait()`
 - `tn_event_wait_polling()`
 - `tn_event_await()`
 - `tn_event_set()`
 - `tn_event_iset()`
 - `tn_event_clear()`
 - `tn_event_iclear()`

Definition at line 341 of file `tn_cfg_default.h`.

20.22.2.13 `#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 ≥ 1 and ≤ 6 . Default: 4.

Definition at line 382 of file `tn_cfg_default.h`.

20.22.2.14 `#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 383 of file `tn_cfg_default.h`.

Index

_TN_TASK_CREATE_OPT_IDLE
tn_tasks.h, 130

TN_CONTEXT_ISR
tn_sys.h, 123

TN_CONTEXT_NONE
tn_sys.h, 123

TN_CONTEXT_TASK
tn_sys.h, 123

TN_EVENTGRP_ATTR_CLR
tn_eventgrp.h, 100

TN_EVENTGRP_ATTR_MULTI
tn_eventgrp.h, 100

TN_EVENTGRP_ATTR_NONE
tn_eventgrp.h, 100

TN_EVENTGRP_ATTR_SINGLE
tn_eventgrp.h, 100

TN_EVENTGRP_OP_CLEAR
tn_eventgrp.h, 100

TN_EVENTGRP_OP_SET
tn_eventgrp.h, 100

TN_EVENTGRP_OP_TOGGLE
tn_eventgrp.h, 100

TN_EVENTGRP_WMODE_AND
tn_eventgrp.h, 99

TN_EVENTGRP_WMODE_AUTOCLR
tn_eventgrp.h, 99

TN_EVENTGRP_WMODE_OR
tn_eventgrp.h, 99

TN_ID_DATAQUEUE
tn_common.h, 91

TN_ID_EVENTGRP
tn_common.h, 91

TN_ID_EXCHANGE
tn_common.h, 91

TN_ID_EXCHANGE_LINK
tn_common.h, 91

TN_ID_FSMEMORYPOOL
tn_common.h, 91

TN_ID_MUTEX
tn_common.h, 91

TN_ID_NONE
tn_common.h, 91

TN_ID_SEMAPHORE
tn_common.h, 91

TN_ID_TASK
tn_common.h, 91

TN_ID_TIMER
tn_common.h, 91

TN_MUTEX_PROT_CEILING

tn_mutex.h, 108

TN_MUTEX_PROT_INHERIT
tn_mutex.h, 108

TN_RC_DELETED
tn_common.h, 92

TN_RC_FORCED
tn_common.h, 92

TN_RC_ILLEGAL_USE
tn_common.h, 92

TN_RC_INTERNAL
tn_common.h, 92

TN_RC_INVALID_OBJ
tn_common.h, 92

TN_RC_OK
tn_common.h, 92

TN_RC_OVERFLOW
tn_common.h, 92

TN_RC_TIMEOUT
tn_common.h, 92

TN_RC_WCONTEXT
tn_common.h, 92

TN_RC_WPARAM
tn_common.h, 92

TN_RC_WSTATE
tn_common.h, 92

TN_STATE_FLAG_DEADLOCK
tn_sys.h, 123

TN_STATE_FLAG_SYS_RUNNING
tn_sys.h, 123

TN_TASK_CREATE_OPT_START
tn_tasks.h, 130

TN_TASK_EXIT_OPT_DELETE
tn_tasks.h, 130

TN_TASK_STATE_DORMANT
tn_tasks.h, 129

TN_TASK_STATE_NONE
tn_tasks.h, 129

TN_TASK_STATE_RUNNABLE
tn_tasks.h, 129

TN_TASK_STATE_SUSPEND
tn_tasks.h, 129

TN_TASK_STATE_WAIT
tn_tasks.h, 129

TN_TASK_STATE_WAITSUSP
tn_tasks.h, 129

TN_WAIT_REASON_DQUE_WRECEIVE
tn_tasks.h, 130

TN_WAIT_REASON_DQUE_WSEND
tn_tasks.h, 130

TN_WAIT_REASON_EVENT
 tn_tasks.h, 129
 TN_WAIT_REASON_MUTEX_C
 tn_tasks.h, 130
 TN_WAIT_REASON_MUTEX_I
 tn_tasks.h, 130
 TN_WAIT_REASON_NONE
 tn_tasks.h, 129
 TN_WAIT_REASON_SEM
 tn_tasks.h, 129
 TN_WAIT_REASON_SLEEP
 tn_tasks.h, 129
 TN_WAIT_REASON_WFIXMEM
 tn_tasks.h, 130
 TN_WAIT_REASONS_CNT
 tn_tasks.h, 130
 tn_common.h
 TN_ID_DATAQUEUE, 91
 TN_ID_EVENTGRP, 91
 TN_ID_EXCHANGE, 91
 TN_ID_EXCHANGE_LINK, 91
 TN_ID_FSMEMORYPOOL, 91
 TN_ID_MUTEX, 91
 TN_ID_NONE, 91
 TN_ID_SEMAPHORE, 91
 TN_ID_TASK, 91
 TN_ID_TIMER, 91
 TN_RC_DELETED, 92
 TN_RC_FORCED, 92
 TN_RC_ILLEGAL_USE, 92
 TN_RC_INTERNAL, 92
 TN_RC_INVALID_OBJ, 92
 TN_RC_OK, 92
 TN_RC_OVERFLOW, 92
 TN_RC_TIMEOUT, 92
 TN_RC_WCONTEXT, 92
 TN_RC_WPARAM, 92
 TN_RC_WSTATE, 92
 tn_eventgrp.h
 TN_EVENTGRP_ATTR_CLR, 100
 TN_EVENTGRP_ATTR_MULTI, 100
 TN_EVENTGRP_ATTR_NONE, 100
 TN_EVENTGRP_ATTR_SINGLE, 100
 TN_EVENTGRP_OP_CLEAR, 100
 TN_EVENTGRP_OP_SET, 100
 TN_EVENTGRP_OP_TOGGLE, 100
 TN_EVENTGRP_WMODE_AND, 99
 TN_EVENTGRP_WMODE_AUTOCLR, 99
 TN_EVENTGRP_WMODE_OR, 99
 tn_mutex.h
 TN_MUTEX_PROT_CEILING, 108
 TN_MUTEX_PROT_INHERIT, 108
 tn_sys.h
 TN_CONTEXT_ISR, 123
 TN_CONTEXT_NONE, 123
 TN_CONTEXT_TASK, 123
 TN_STATE_FLAG_DEADLOCK, 123
 TN_STATE_FLAG_SYS_RUNNING, 123
 tn_tasks.h
 _TN_TASK_CREATE_OPT_IDLE, 130
 TN_TASK_CREATE_OPT_START, 130
 TN_TASK_EXIT_OPT_DELETE, 130
 TN_TASK_STATE_DORMANT, 129
 TN_TASK_STATE_NONE, 129
 TN_TASK_STATE_RUNNABLE, 129
 TN_TASK_STATE_SUSPEND, 129
 TN_TASK_STATE_WAIT, 129
 TN_TASK_STATE_WAITSUSP, 129
 TN_WAIT_REASON_DQUE_WRECEIVE, 130
 TN_WAIT_REASON_DQUE_WSEND, 130
 TN_WAIT_REASON_EVENT, 129
 TN_WAIT_REASON_MUTEX_C, 130
 TN_WAIT_REASON_MUTEX_I, 130
 TN_WAIT_REASON_NONE, 129
 TN_WAIT_REASON_SEM, 129
 TN_WAIT_REASON_SLEEP, 129
 TN_WAIT_REASON_WFIXMEM, 130
 TN_WAIT_REASONS_CNT, 130