

Исходным материалом для данного файла послужила документация, размещенная по адресу <http://www.pic24.ru/doku.php/tkernel/ref/intro> по состоянию на 23.01.2011. Цель: создать документ, пригодный для печати на принтере.

TNKernel

Document Disclaimer

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

Copyright notice

TNKernel real time kernel Copyright © 2004,2006 Yuri Tiomkin All rights reserved. Permission to use, copy, modify, and distribute this software in source and binary forms and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. THIS SOFTWARE IS PROVIDED BY THE YURI TIOMKIN AND CONTRIBUTORS ``AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL YURI TIOMKIN OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Trademarks

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

Document Disclaimer	1
Copyright notice.....	1
Trademarks	1
Введение	5
1.TNKernel : Задачи	6
1.1. Введение	6
1.2. Состояния задач	6
1.3. Правила планирования	8
1.4. Системные задачи	8
1.5. Структура управления задачей	8
1.6. Сервисы управления задачами	10
2.TNKernel : Семафоры	12
2.1. Введение	12
2.2. Структура управления семафором.....	12
2.3. Сервисы управления семафорами	13
3.TNKernel : Флаги	14
3.1. Введение	14
3.2. Структура управления флагом	14
3.3. Сервисы управления флагами	14
4.TNKernel : Очереди сообщений	16
4.1. Введение	16
4.2. Структура управления очередью сообщений.....	17
4.3. Сервисы управления очередями сообщений	18
5.TNKernel : Мьютексы	19
5.1. Введение	19
5.2. Инверсия приоритетов	19
5.2.1. Протокол наследования приоритета	20
5.2.2. Протокол увеличения приоритета	20
5.3. Взаимная блокировка.....	21
5.4. Структура управления мьютексом.....	22
5.5. Сервисы управления мьютексами	23
6.TNKernel : Блоки памяти фиксированного размера	24
6.1. Введение	24
6.2. Структура управления пулом блоков памяти	24
6.3. Сервисы управления пулами блоков памяти	25
7.TNKernel : Системные сервисы.....	26
7.1. Введение	26
7.2. Запуск системы	26
7.3. Системный таймер.....	27
7.4. Управление Round-Robin	27
7.5. Запрещение переключения контекста	27
7.6. Системное время	28
7.7. Системные сервисы	29

8.Отличия TNKernel для PIC24/dsPIC и PIC32	30
8.1. Основные отличия от оригинальной версии	31
8.1.1. Типы данных	31
8.1.2. Приоритеты задач	32
8.1.3. Инициализация системы	32
8.1.4. Создание задачи	33
8.2. Нововведения	34
8.2.1. 1. Критические секции	34
8.2.2. Новые сервисы	34
8.2.3. Атрибут задачи	34
8.2.4. Атрибут данных	34
8.2.5. Отладка	35
8.2.6. Варианты сервисов без проверки параметров	35
8.2.7. Контроль переполнения стеков задач	35
8.2.8. Код возврата TERR_EXS	36
8.2.9. Получение ревизии TNKernel	36
8.2.10. Системный таймер	37
8.2.11. Файл конфигурации	37
8.3. Использование прерываний	38
8.4. Отличия порта для PIC32 от порта для PIC24/dsPIC	39
9.Сервисы RTOS	40
9.1. Сервисы управления задачами	40
9.1.1. Создание и удаление задачи	40
9.1.2. Перезапуск задачи	43
9.1.3. Останов и восстановление задачи	47
9.1.4. Приостановка выполнения и пробуждение задачи	51
9.1.5. Форсированный вывод задачи из состояния WAITING	54
9.1.6. Изменение приоритета задачи	56
9.1.7. Получение информации о задаче	57
9.2. Сервисы управления семафорами	61
9.2.1. Создание и удаление семафора	61
9.2.2. Освобождение семафора	63
9.2.3. Захват семафора	65
9.3. Сервисы управления флагами	68
9.3.1. Создание и удаление флага	68
9.3.2. Установка и сброс битовой маски флага	70
9.3.3. Ожидание флага	74
9.4. Сервисы управления очередями сообщений	80
9.4.1. Создание и удаление очереди сообщений	80
9.4.2. Отсылка сообщения	82
9.4.3. Прием сообщения	85
9.5. Сервисы управления мютексами	88
9.5.1. Создание и удаление мютекса	88
9.5.2. Блокировка мютекса	90
9.5.3. Освобождение мютекса	92
9.6. Сервисы управления пулами блоков памяти	93

9.6.1. Создание и удаление пула	93
9.6.2. Получение блока памяти	95
9.6.3. Освобождение блока памяти	98
9.7. Системные сервисы	100
9.7.1. Основные сервисы	100
9.7.2. Запрещение переключения контекста	105
9.7.3. Системное время	107

INDEX

tn_event_clear()	15, 72	tn_sem_ipolling()	13, 67
tn_event_create()	15, 36, 68	tn_sem_isignal()	13, 64
tn_event_delete()	15, 69	tn_sem_polling()	13, 66
tn_event_iclear()	15, 73	tn_sem_signal()	13, 63
tn_event_iset()	15, 71	tn_start_system()	26, 29, 32, 33, 38, 100
tn_event_await()	15, 76	tn_sys_context_get()	29, 34, 104
tn_event_set()	15, 70	tn_sys_enter_critical()	26, 27, 28, 29, 34, 39, 105, 106
tn_event_wait()	15, 74	tn_sys_exit_critical()	26, 27, 28, 29, 34, 39, 105, 106
tn_event_wait_polling()	15, 78	tn_sys_time_get()	26, 28, 29, 37, 107, 108
tn_fmem_create()	25, 36, 93	tn_sys_time_set()	108
tn_fmem_delete()	25, 94	tn_sys_tslice_ticks()	26, 27, 29, 103
tn_fmem_get()	25, 95	tn_task_activate()	10, 43, 44, 45
tn_fmem_get_ipolling()	25, 97	tn_task_change_priority()	11, 56
tn_fmem_get_polling()	25, 96	tn_task_create()	10, 33, 36, 40
tn_fmem_irelease()	25, 99	tn_task_delete()	10, 42
tn_fmem_release()	25, 98	tn_task_exit()	10, 43, 44
tn_mutex_create()	23, 36, 88	tn_task_iactivate()	10, 43, 44, 46
tn_mutex_delete()	23, 89	tn_task_ireference()	11, 34, 59
tn_mutex_lock()	23, 90	tn_task_irelease_wait()	11, 55
tn_mutex_lock_polling()	23, 91	tn_task_iresume()	11, 34, 50
tn_mutex_unlock()	23, 92	tn_task_isuspend()	11, 34, 48
tn_queue_create()	18, 36, 80	tn_task_iwakeup()	11, 51, 53
tn_queue_delete()	18, 81	tn_task_reference()	11, 34, 57
tn_queue_ireceive()	18, 87	tn_task_release_wait()	11, 54
tn_queue_isend_polling()	18, 84	tn_task_resume()	11, 49
tn_queue_receive()	18, 85	tn_task_sleep()	11, 51, 52, 53, 54, 55, 57, 59
tn_queue_receive_polling()	18, 86	tn_task_suspend()	10, 47
tn_queue_send()	18, 82	tn_task_terminate()	10, 43, 44
tn_queue_send_polling()	18, 83	tn_task_wakeup()	11, 51, 52
tn_sem_acquire()	13, 65	tn_tick_int_processing()	26, 27, 29, 39, 102
tn_sem_create()	13, 36, 61		
tn_sem_delete()	13, 62		

Введение

TNKernel - компактная и быстрая операционная система реального времени, разработанная для 32- и 16-битных однокристальных микроконтроллеров. TNKernel обеспечивает вытесняющее планирование, основанное на приоритетах задач и карусельное переключение задач с равными приоритетами. В основу TNKernel положена спецификация μ ITRON 4.0¹⁾

¹⁾ μ ITRON 4.0 - это открытая спецификация, описывающая операционную систему реального времени, разработанная комитетом ITRON ассоциации TRON.

Текущая версия TNKernel поддерживает следующие объекты:

- задачи
- семафоры
- мютексы
- очереди сообщений
- флаги
- блоки памяти фиксированного размера

Большинство кода TNKernel написано на языке Си, что позволило портировать оригинальную версию, которая изначально была разработана для микроконтроллеров с ядром ARM7, на 16-битную архитектуру Microchip PIC24/dsPIC.

Операционная система TNKernel распространяется в открытых исходных кодах под лицензией FreeBSD-like.

Документация учитывает изменения и дополнения, внесенные в порт TNKernel для PIC24/dsPIC

1. TNKernel : Задачи

1.1. Введение

Задача в TNKernel это часть программного кода, которая с точки зрения программиста выполняется *одновременно* с другими задачами, что обеспечивается разделением процессорного времени между ними. Каждая задача может быть представлена как независимое приложение, которое владеет уникальными ресурсами (регистры процессора, указатель стека и т.п.). Эти ресурсы называются контекстом задачи, а время в течении которого задача выполняется можно назвать временем в *контексте задачи*.

Когда текущая задача приостанавливает выполнение (в случае прерывания или вызова сервиса), осуществляется *переключение контекста* - контекст текущей задачи сохраняется в ее стеке, а контекст наиболее приоритетной задачи из готовых к выполнению восстанавливается. Этот механизм в TNKernel называется "диспетчером".

Определение наиболее приоритетной задачи в момент переключения контекста осуществляется на основании набора правил, а механизм, который обеспечивает соблюдение этих правил называется "планировщиком".

В TNKernel используется приоритетное вытесняющее планирование, основанное на приоритете, назначаемом каждой задаче, при этом чем меньше величина, тем выше уровень приоритета. В TNKernel доступно 32 уровня приоритета для 32-битных контроллеров (ARM, MIPS) и 16 уровней приоритета для 16-битных контроллеров (PIC24/dsPIC).

Приоритеты 0 (самый высокий) и 31(**15**) (самый низкий) зарезервированы для системных задач. Для пользовательских задач доступны приоритеты от 1 до 30(**14**) включительно. В TNKernel несколько задач могут иметь одинаковый приоритет.

1.2. Состояния задач

Задачи в TNKernel могут находиться в одном из четырех состояний:

RUNNING

Задача выполняется в данный момент

READY

Задача готова к выполнению, но не может получить , так как в данный момент выполняется задача с более высоким (или равным) приоритетом. В TNKernel оба состояния RUNNING и READY называются RUNNABLE

WAIT/SUSPEND

Когда задача находится в состоянии WAIT/SUSPEND она не может начать выполнение до тех пор пока не выполнится условие, которого задача ожидает. При входе в состояние WAIT/SUSPEND контекст задачи сохраняется, при выходе из этого состояния контекст восстанавливается. Состояние WAIT/SUSPEND делится на три типа:

WAITING

Задача находится в состоянии WAIT/SUSPEND до тех пор пока не наступит событие, которого она ожидает - завершится таймаут, освободится семафор, установится флаг и т.п.

SUSPENDED

Задача перемещена в состояние WAIT/SUSPEND другой задачей или самостоятельно путем вызова специального сервиса

WAITING_SUSPENDED

Задача находится как в состоянии `WAITING`, так и в состоянии `SUSPENDED` (ожидает события и приостановлена специальным сервисом). Если задача освобождается от состояния `WAITING` (ожидаемое событие наступило), то она остается в состоянии `SUSPENDED` и наоборот.

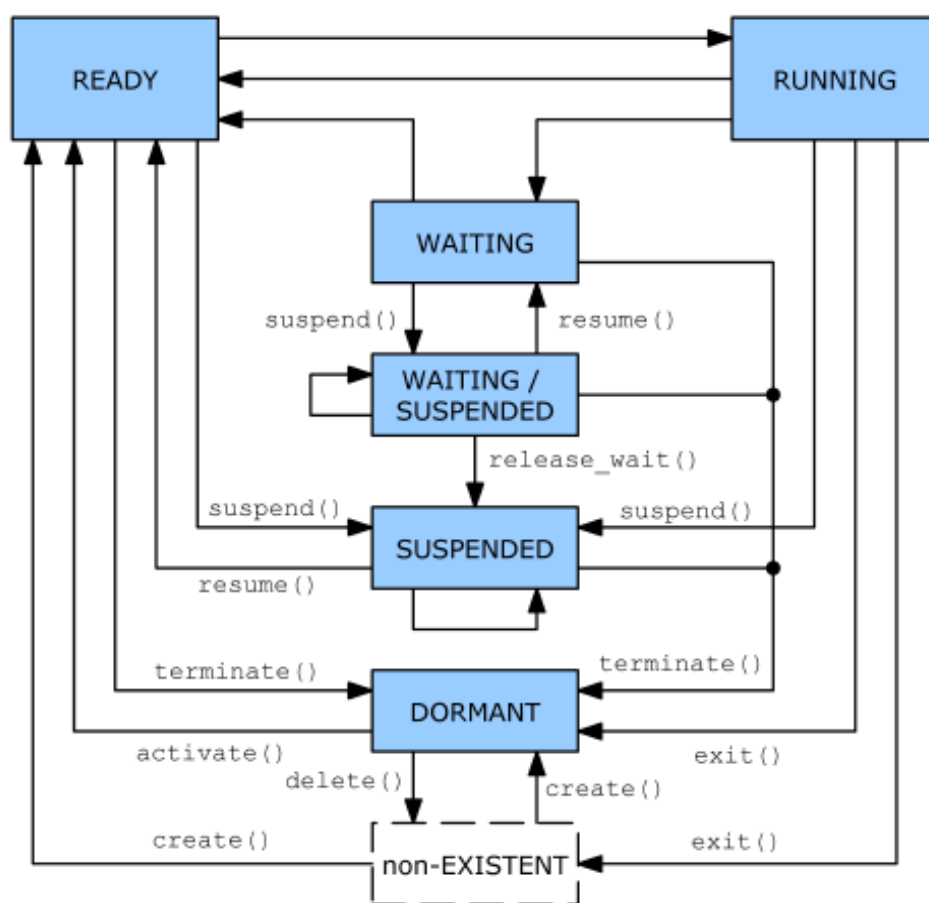
DORMANT

Задача уже создана, но еще ни разу не запускалась

Выполнение задачи завершено с помощью специального таймера.

Можно так же выделить состояние задачи, в котором она еще не создана - состояние `NON-EXISTENT`.

Граф перехода между состояниями изображен на рисунке:



Сервисы, вызов которых приводит к изменению состояния задачи указаны возле направлений перехода. Для простоты префикс `tn_task_` и префикс `i` (вызов из прерывания) опущены.

Переход задачи из состояния `READY` в состояние `RUNNING` происходит тогда, когда ее приоритет наивысший из приоритетов всех задач находящихся в состоянии `READY`.

Переход задачи из состояния `RUNNING` в состояние `READY` происходит тогда, когда ее приоритет меньше чем приоритет одной из задач, находящихся в состоянии `READY`.

Задача переходит из состояния `RUNNING` в состояние `WAITING` при вызове любого из системных сервисов, блокирующих выполнение задачи. Например, если задача пытается захватить семафор, а он занят другой задачей, ей ничего не остается как ожидать его освобождения в состоянии `WAITING`.

Задача переходит из состояния `WAITING` в состояние `READY` когда событие, которого она ожидала произошло. Например, освободился семафор или окончился таймаут ожидания.

1.3. Правила планирования

В TNKernel во время выполнения задачи с наивысшим приоритетом ни одна из других задач не может получить управление до тех пор, пока эта задача не перейдет в состояние WAITING/SUSPEND или DORMANT.

Если несколько задач с *разными приоритетами* готовы к выполнению (т.е. находятся в состоянии READY), управление получит задача с наивысшим приоритетом.

Если несколько задач с *одинаковым приоритетом* готовы к выполнению, то управление получит задача, которая перешла в состояние READY раньше остальных, т.е. первой стоит в очереди готовых к выполнению.

Пример: пусть задача **А** имеет приоритет 1, задачи **Б, В, Г, Д** - приоритет 3, задачи **Е и Ж** - приоритет 4, задача **З** - приоритет 5. Если все задачи находятся в состоянии READY последовательность выполнения будет следующая: 1. Задача **А** с наиболее высоким приоритетом (приоритет 1) 2. Задачи **Б, В, Г и Д** в той последовательности в которой они перешли в состояние READY (приоритет 3) 3. Задачи **Е и Ж** в той последовательности, в которой они перешли в состояние READY (приоритет 4) 4. Задача **З**, так как она имеет наиболее низкий приоритет (приоритет 5)

В TNKernel задачи с одинаковым приоритетом могут получать управление в соответствии с правилами round-robin планировщика (планировщика карусельного типа). В этом случае каждой задаче выделяется квант времени. Квант времени может быть выбран для каждого приоритета.

1.4. Системные задачи

В TNKernel существует две системные задачи, которые создаются при запуске системы.

Одна из этих задач (timer_task) имеет наивысший приоритет (0) и обеспечивает функционирование системного таймера.

Вторая задача (idle_task) имеет минимальный приоритет (31/15) и получает управление тогда, когда нет пользовательских задач готовых к выполнению. Эта задача может использоваться для сбора статистики или перевода процессора в состояние пониженного потребления энергии.

Размеры стеков системных задач определяет пользователь, исходя из особенностей приложения. Кроме того, необходимо объявить функцию idle_user_cb(), которая циклически вызывается из задачи idle_task.

1.5. Структура управления задачами

Каждая задача ассоциируется со структурой управления:

```
typedef struct TN_TCB_S_STRUCT
{
    TN_UWORD          * task_stk;
    CDLL_QUEUE_S      task_queue;
    CDLL_QUEUE_S      timer_queue;
    CDLL_QUEUE_S      block_queue;
    CDLL_QUEUE_S      create_queue;
    CDLL_QUEUE_S      mutex_queue;
    CDLL_QUEUE_S      * pwait_queue;
    struct TN_TCB_S_STRUCT * blk_task;

    TN_UWORD          * stk_start;
    TN_UWORD          stk_size;

    void              * task_func_addr;
    void              * task_func_param;

    TN_UWORD          base_priority;
    TN_UWORD          priority;
    TN_UWORD          id_task;
```



```

    TN_WORD      task_state;
    TN_UWORD     task_wait_reason;
    TN_WORD      task_wait_rc;
    TN_UWORD     tick_count;
    TN_UWORD     tslice_count;

    TN_UWORD     ewart_pattern;
    TN_UWORD     ewart_mode;

    void          *data_elem;

    TN_UWORD     activate_count;
    TN_UWORD     wakeup_count;
    TN_UWORD     suspend_count;
} TN_TCB_S;

```

В состав структуры управления задачами входят следующие элементы:

<code>task_stk</code>	Указатель на вершину стека задачи
<code>task_queue</code>	Элемент очереди для включения задачи в список существующих задач
<code>timer_queue</code>	Элемент очереди для включения задачи в список задач, ожидающих события таймера (таймаут и т.п.)
<code>block_queue</code>	Элемент очереди для включения задачи в список заблокированных задач (используется в протоколе priority ceiling системных мютексов)
<code>create_queue</code>	Элемент очереди для включения задачи в список созданных задач
<code>mutex_queue</code>	Список всех мютексов, заблокированных задачами
<code>pwait_queue</code>	Указатель на очередь объектов (семафоров, флагов), ожидаемых задачами
<code>blk_task</code>	Указатель на структуру задачи которая заблокировала эту задачу (используется в протоколе priority ceiling системных мютексов)
<code>stk_start</code>	Указатель на базовый адрес стека задачи
<code>stk_size</code>	Размер стека задачи
<code>task_func_addr</code>	Указатель на функцию задачи
<code>task_func_param</code>	Указатель на параметр, передаваемый в функцию задачи
<code>base_priority</code>	Базовый приоритет задачи
<code>priority</code>	Текущий приоритет задачи
<code>id_task</code>	Поле идентификации объекта как задачи
<code>task_state</code>	Состояние задачи
<code>task_wait_reason</code>	Причина нахождения в состоянии WAITING
<code>task_wait_rc</code>	Код, возвращаемый задачей при выходе из состояния WAITING (причина, по

	которой задача вышла из состояния ожидания)
<code>tick_count</code>	Время до истечения таймаута в системных тиках
<code>tslice_count</code>	Счетчик кванта времени для round-robin планирования в системных тиках
<code>ewait_pattern</code>	Маска ожидаемых флагов
<code>ewait_mode</code>	Тип ожидания флагов (И или ИЛИ, т.е. все флаги, или хотя бы один из маски)
<code>data_elem</code>	Указатель на очередь сообщений
<code>activate_count</code>	Счетчик запросов на активацию задачи
<code>wakeup_count</code>	Счетчик запросов на пробуждение задачи
<code>suspend_count</code>	Счетчик запросов на останов задачи



Структура задачи доступна только при определении `TN_DEBUG`. Тем не менее, прямой доступ к элементам структуры задачи крайне не рекомендуется, так как это является вмешательством в работу планировщика и других сервисов RTOS.

1.6. Сервисы управления задачами

TNKernel имеет следующий набор функций (сервисов) для управления задачами:

Сервис	Описание	Свойства
--------	----------	----------

Создание и удаление задачи

<code>tn_task_create()</code>	Создание задачи
---	-----------------



<code>tn_task_delete()</code>	Удаление задачи
---	-----------------



Перезапуск задачи

<code>tn_task_exit()</code>	Выход из текущей задачи
---	-------------------------



<code>tn_task_terminate()</code>	Завершение работы задачи
--	--------------------------



<code>tn_task_activate()</code>	Перезапуск задачи
---	-------------------









<code>tn_task_iactivate()</code>	Перезапуск задачи из прерывания
--	---------------------------------









Останов и восстановление задачи

<code>tn_task_suspend()</code>	Останов задачи
--	----------------







tn_task_isuspend()	Останов задачи в прерывании	 
tn_task_resume()	Восстановление задачи	 
tn_task_iresume()	Восстановление задачи в прерывании	 



Приостановка выполнения и пробуждение задачи

tn_task_sleep()	Приостановка выполнения задачи	 
tn_task_wakeup()	Пробуждение приостановленной задачи	 
tn_task_iwakeup()	Пробуждение приостановленной задачи в прерывании	 



Форсированный вывод задачи из состояния WAITING

tn_task_release_wait()	Форсированный вывод задачи из состояния WAITING	 
tn_task_irelease_wait()	Форсированный вывод задачи из состояния WAITING в прерывании	 

Изменение приоритета задачи

tn_task_change_priority()	Изменение приоритета задачи	 
---	-----------------------------	---

Получение информации о задаче

tn_task_reference()	Получение информации о задаче	
tn_task_ireference()	Получение информации о задаче в прерывании	

2. TNKernel : Семафоры

2.1. Введение

Семафор - это объект RTOS, предназначенный для синхронизации задач и обеспечения конкурентного доступа к ресурсам программы. В текущий момент времени только одна задача может "владеть" семафором. Задача, пытающаяся завладеть уже захваченным семафором переводится в состояние `WAITING` и ставится в очередь задач, ожидающих семафор, до тех пор пока он не будет освобожден.

Существует несколько типов семафоров - счетный семафор, двоичный семафор (подтип счетного) и семафор конкурентного доступа или [мьютекс](#). В TNKernel мьютекс является отдельным объектом и рассмотрен в соответствующем [разделе](#).

Счетный семафор можно сравнить с менеджером отеля у которого есть несколько ключей от одной комнаты. Менеджер будет выдавать жильцам комнаты (задачам RTOS) ключи до тех пор, пока они не кончатся (семафор захвачен). Как только один из жильцов вернет ключ от комнаты, он сразу будет выдан жильцу (задаче), ожидающему ключ.

Таким образом со счетным семафором ассоциировано такое понятие как *счетчик свободных ресурсов*. Если счетчик больше нуля - семафор может быть захвачен одной из задач, при этом счетчик ресурсов уменьшается на единицу. Если счетчик равен нулю - семафор заблокирован, и все задачи пытающиеся его захватить будут поставлены в очередь ожидания семафора. Как только счетчик ресурсов семафора станет больше нуля - семафор захватит первая задача из очереди ожидающих.

Двоичный семафор является подтипом счетного - его счетчик свободных ресурсов может принимать только значения 0 (захвачен) и 1 (свободен). В TNKernel семафоры не разделяются на двоичные и счетные - максимальное значение счетчика ресурсов задается при создании семафора.

В TNKernel освободившийся семафор захватит задача, которая стоит первой в очереди задач ожидающих этот семафор, вне зависимости от приоритетов задач в очереди.

2.2. Структура управления семафором

Каждый семафор ассоциируется со структурой управления:

```
typedef struct TN_SEM_S_STRUCT
{
    CDLL_QUEUE_S    wait_queue;
    TN_UWORD        count;
    TN_UWORD        max_count;
    TN_OBJ_ID       id_sem;
} TN_SEM_S;
```

В состав структуры семафора входят следующие элементы:

`wait_queue` Очередь задач, ожидающих освобождения семафора

`count` Счетчик свободных ресурсов семафора

`max_count` Максимальное значение счетчика свободных ресурсов

`id_sem` Поле идентификации объекта как семафора



Структура семафора доступна только при определении `TN_DEBUG`. Тем не менее, прямой доступ к элементам структуры семафора крайне не рекомендуется, так как это является вмешательством в работу планировщика и других сервисов RTOS.

2.3. Сервисы управления семафорами

TNKernel имеет следующий набор функций (сервисов) для управления семафорами:

Сервис	Описание	Свойства
--------	----------	----------

Создание и удаление семафора

tn_sem_create()	Создание семафора
---------------------------------	-------------------



tn_sem_delete()	Удаление семафора
---------------------------------	-------------------



Освобождение семафора

tn_sem_signal()	Освобождение семафора
---------------------------------	-----------------------



tn_sem_issignal()	Освобождение семафора в прерывании
-----------------------------------	------------------------------------



Захват семафора

tn_sem_acquire()	Захват семафора
----------------------------------	-----------------



tn_sem_polling()	Попытка захвата семафора без блокировки задачи
----------------------------------	--



tn_sem_ipolling()	Попытка захвата семафора в прерывании
-----------------------------------	---------------------------------------



3. TNKernel : Флаги

3.1. Введение

Флаг, как и [семафор](#), является объектом RTOS, предназначенным для синхронизации задач. В отличие от семафора флаг не имеет счетчика свободных ресурсов, однако с каждым флагом ассоциирован элемент, называемый *битовой маской*. Битовая маска это переменная с разрядностью, равной, как правило, разрядности машинного слова. В TNKernel битовая маска флага имеет разрядность 16 бит для PIC24/dsPIC и 32 бита для ARM/PIC32.

Любая задача (или системное прерывание) может с помощью [сервисов управления флагом](#) установить или сбросить определенные биты в битовой маске, сигнализируя таким образом об определенном событии.

Задача может ожидать появления определенного набора битов в битовой маске. Как только битовая маска флага станет равна ожидаемой, задача перейдет в состояние готовых к выполнению.

В TNKernel существует два типа флагов. Первый тип подразумевает, что ожидать события (то есть определенного набора битов в маске флага) будет несколько задач. Второй тип разрешает прием события только одной задаче - в этом случае биты в маске могут быть сброшены средствами сервиса.

В сервис ожидания в качестве параметра передается аргумент, который указывает, какая логика будет использоваться для ожидания битовой маски: AND или OR. В первом случае задача будет переведена в состояние готовых к выполнению, если *все* ожидаемые биты будут установлены. Во втором случае - если *хотя бы один* из ожидаемых битов будет установлен.



В TNKernel будет запущена задача, которая стоит первой в очереди задач ожидающих флаг, вне зависимости от приоритетов задач в очереди.

3.2. Структура управления флагом

Каждый флаг ассоциируется со структурой управления:

```
typedef struct _TN_EVENT_S
{
    CDLL_QUEUE_S    wait_queue;
    TN_UWORD        attr;
    TN_UWORD        pattern;
    TN_OBJ_ID       id_event;
} TN_EVENT_S;
```

В состав структуры флага входят следующие элементы:

`wait_queue` Очередь задач, ожидающих флаг

`attr` Тип флага - для всех задач или для одной задачи

`pattern` Битовая маска флага






`id_event` Поле идентификации объекта как флага



Структура флага доступна только при определении `TN_DEBUG`. Тем не менее, прямой доступ к элементам структуры флага крайне не рекомендуется, так как это является вмешательством в работу планировщика и других сервисов RTOS.

3.3. Сервисы управления флагами

TNKernel имеет следующий набор функций (сервисов) для управления флагами:

Сервис	Описание	Свойства
Создание и удаление флага		
<u>tn_event_create()</u>	Создание флага	
<u>tn_event_delete()</u>	Удаление флага	
Установка и сброс битовой маски флага		
<u>tn_event_set()</u>	Установка битов в битовой маске	
<u>tn_event_iset()</u>	Установка битов в битовой маске в прерывании	
<u>tn_event_clear()</u>	Сброс битов в битовой маске	
<u>tn_event_iclear()</u>	Сброс битов в битовой маске в прерывании	
Ожидание флага		
<u>tn_event_wait()</u>	Ожидание флага	
<u>tn_event_await()</u>	Ожидание флага в прерывании	
<u>tn_event_wait_polling()</u>	Ожидание флага без блокировки	

4. TNKernel : Очереди сообщений

4.1. Введение

Очередь сообщений - это объект RTOS, предназначенный для передачи данных между задачами и между прерываниями и задачами.

В **не** RTOS системе передача данных между различными подпрограммами, которые вызываются в бесконечном цикле (условно их можно назвать задачами), происходит, как правило, с помощью глобальных переменных.

Однако в многозадачной системе использование глобальных переменных крайне не рекомендуется - так как вытеснение может произойти в любой момент, необходимо реализовать для доступа к глобальной переменной критическую секцию, что может привести к значительной трате ресурсов. Поэтому передача данных в RTOS как правило реализуется с помощью специальных объектов - очередей сообщений.

Кроме защиты сообщений, объект реализует *очередь*, то есть возможность передать несколько сообщений, которые могут быть обработаны принимающей стороной позднее.

Очередь сообщений ассоциируется со структурой управления и буфером сообщений. Буфер является очередью типа FIFO, т.е. первым будет получено сообщение, которое отправлено раньше.

В TNKernel под сообщением (т.е. элементом, который хранится в буфере) понимается **указатель на данные**, а не сами данные. Другими словами - сообщение это не данные, а указатель на данные. Такой подход имеет как плюсы, так и минусы.

Плюсы:

объем передаваемых данных может быть сколь угодно большим - в качестве сообщения можно передавать указатель на структуру или массив

маленький размер буфера, так как в нем хранятся только указатели на данные

Минусы:

данные по передаваемому указателю не могут быть изменены до тех пор, пока сообщение не будет принято и обработано. Это можно обойти, используя пересылку с подтверждением, либо блоки памяти фиксированного размера.

Тем не менее, в качестве сообщения можно передавать не указатель, а например, код команды - для этого необходимо явно привести типы при передаче параметров в сервисы отсылки и приема сообщения.

Основными сервисами управления очередями сообщений являются отсылка сообщения и прием сообщения.

Если задача вызывает сервис приема сообщения, а в буфер очереди пуст, задача переводится в состояние ожидания, до тех пор, пока сообщение не будет отправлено другой задачей или пока не истечет таймаут.



В TNKernel будет запущена задача, которая стоит первой в очереди задач ожидающих сообщение, вне зависимости от приоритетов задач в очереди.

Если задача вызывает сервис передачи сообщения, а буфер сообщения полон, то она переводится в состояние ожидания до тех пор, пока хотя бы одно сообщение не будет принято.

4.2. Структура управления очередью сообщений

Каждая очередь сообщений ассоциируется со структурой управления:

```
typedef struct _TN_DQUE_S
{
    CDLL_QUEUE_S    wait_send_list;
    CDLL_QUEUE_S    wait_receive_list;

    void            ** data_fifo;
    TN_UWORD         num_entries;
    TN_UWORD         tail_cnt;
    TN_UWORD         header_cnt;
    TN_OBJ_ID        id_dque;
} TN_DQUE_S;
```

В состав структуры очереди входят следующие элементы:

`wait_send_list` Очередь задач, посылающих сообщение

`wait_receive_list` Очередь задач, принимающих сообщение

`data_fifo` Указатель на буфер сообщений. Буфер сообщений это массив указателей на `void`

`num_entries` Объем буфера (максимальное количество сообщений в очереди)

`tail_cnt` Индекс принимаемого сообщения

`header_cnt` Индекс отсылаемого сообщения

`id_dque` Поле идентификации объекта как очереди сообщений



Структура очереди сообщений доступна только при определении `TN_DEBUG`. Тем не менее, прямой доступ к элементам структуры крайне не рекомендуется, так как это является вмешательством в работу планировщика и других сервисов RTOS.

4.3. Сервисы управления очередями сообщений

TNKernel имеет следующий набор функций (сервисов) для управления очередями сообщений:

Сервис	Описание	Свойства
Создание и удаление очереди сообщений		
<u>tn_queue_create()</u>	Создание очереди	A blue circle with a yellow 't' and a purple 'i'.
<u>tn_queue_delete()</u>	Удаление очереди	A blue circle with a yellow 't' and a blue circle with a yellow 'H'.
Отсылка сообщения		
<u>tn_queue_send()</u>	Отсылка сообщения	A blue circle with a yellow 't', a blue circle with a yellow 'H', and a yellow circle with a blue 'S'.
<u>tn_queue_send_polling()</u>	Отсылка сообщения без блокировки задачи	A blue circle with a yellow 't' and a blue circle with a yellow 'H'.
<u>tn_queue_isend_polling()</u>	Отсылка сообщения из прерывания	A purple circle with a yellow 'i' and a blue circle with a yellow 'H'.
Прием сообщения		
<u>tn_queue_receive()</u>	Прием сообщения	A blue circle with a yellow 't', a blue circle with a yellow 'H', and a yellow circle with a blue 'S'.
<u>tn_queue_receive_polling()</u>	Прием сообщения без блокировки задачи	A blue circle with a yellow 't' and a blue circle with a yellow 'H'.
<u>tn_queue_ireceive()</u>	Прием сообщения в прерывании	A purple circle with a yellow 'i' and a blue circle with a yellow 'H'.

5. TNKernel : Мьютексы

5.1. Введение

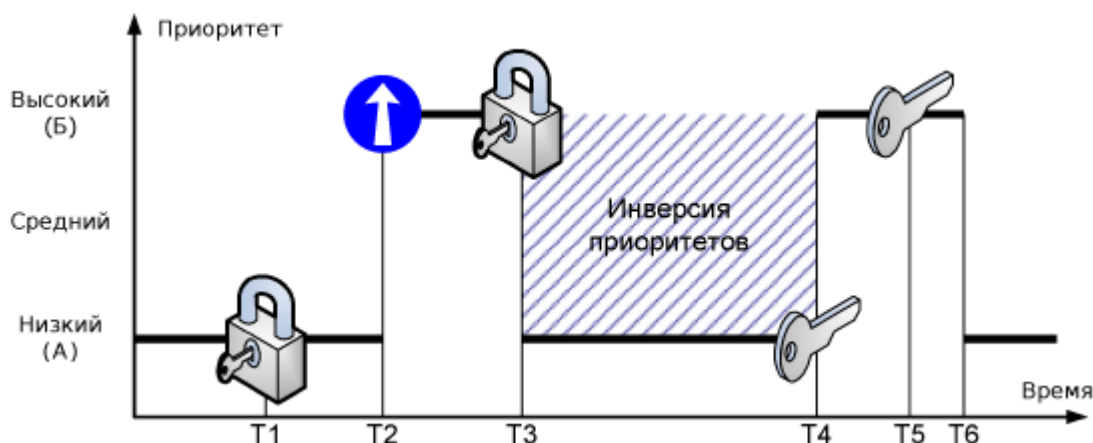
Мьютекс - это объект RTOS, предназначенный для обеспечения конкурентного доступа к общим ресурсам. Мьютекс представляет собой двоичный [семафор](#) с дополнительными свойствами (например, протоколы обхода неограниченной инверсии приоритетов).

Мьютекс может находиться в двух состояниях: заблокированном и разблокированном. Ассоциировав мьютекс с аппаратным или программным ресурсом приложения, можно обеспечить корректный доступ к ресурсам из нескольких задач - если задача попытается получить доступ к заблокированному ресурсу, она будет переведена в состояние ожидания, но получит управление сразу же после того как ресурс будет разблокирован.

Часто возникает вопрос - а зачем вообще нужно блокировать ресурсы? Все дело в принципе работы вытесняющих RTOS - задача может быть прервана (вытеснена) в любой момент времени. Если в этот момент она использует некий системный ресурс (например, UART), а задача, которая вытеснила текущую так же начнет с ним работать - возникнет закономерный конфликт.

5.2. Инверсия приоритетов

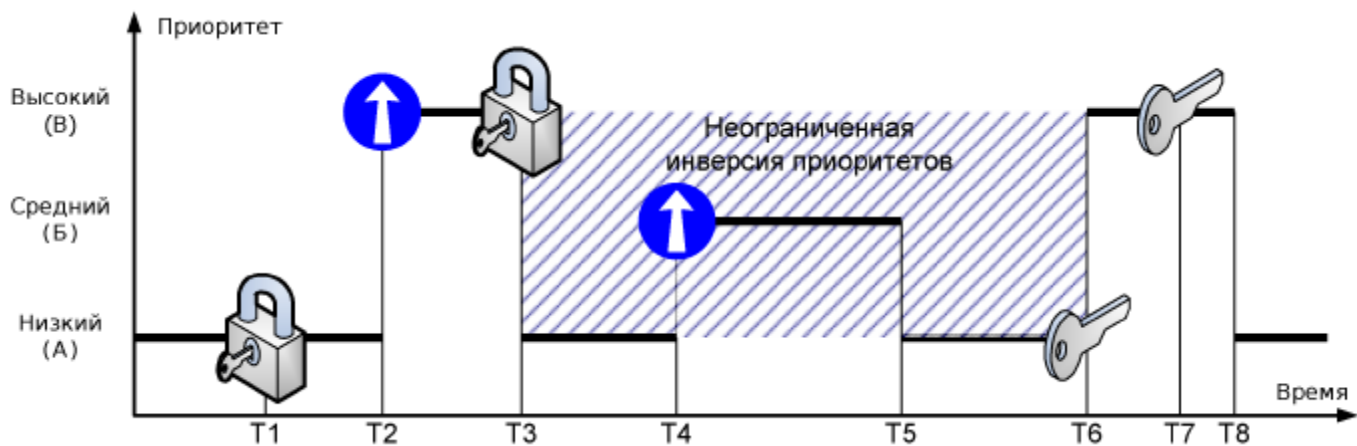
С блокировкой ресурсов тесно связано понятие инверсии приоритетов:



Допустим в системе существуют две задачи с низким (А) и высоким (Б) приоритетом. В момент времени T1 задача (А) блокирует ресурс и начинает его обслуживать. В момент времени T2 задача (Б) вытесняет низкоприоритетную задачу (А) и пытается завладеть ресурсом в момент времени T3. Но так как ресурс заблокирован, задача (Б) переводится в ожидание, а задача (А) продолжает выполнение. В момент времени T4 задача (А) завершает обслуживание ресурса и разблокирует его. Так как ресурс ожидает задача (Б), она тут же начинает выполнение.

Временной промежуток (T4-T3) называют **ограниченной инверсией приоритетов**. В этом промежутке наблюдается логическое несоответствие с правилами планирования - задача с более высоким приоритетом находится в ожидании в то время как низкоприоритетная задача выполняется.

Но это еще не самое страшное. Допустим в системе работают три задачи: низкоприоритетная (А), со средними приоритетом (Б) и высокоприоритетная (В):



Если ресурс заблокирован задачей (А), а он требуется задаче (В), то наблюдается та же ситуация - высокоприоритетная задача блокируется. Но допустим, что задача (В) вытеснила (А), после того как (В) ушла в ожидание ресурса. Задача (В) ничего не знает о конфликте, поэтому может выполняться сколь угодно долго на промежутке времени (Т5-Т4). Кроме того, кроме (В) в системе могут быть и другие задачи, с приоритетами больше (А), но меньше (В). Поэтому длительность периода (Т6-Т3) в общем случае неопределена. Такую ситуацию называют **неограниченной инверсией приоритетов**.

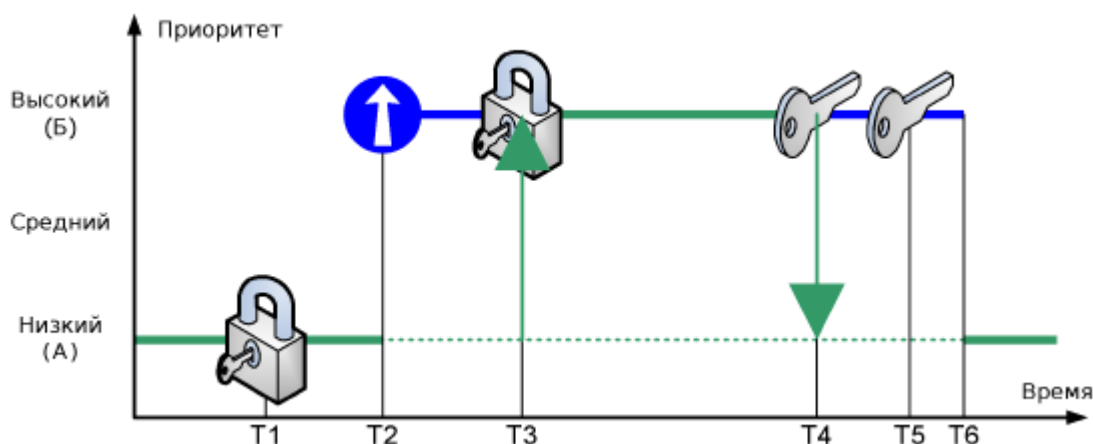
Ограниченной инверсии приоритетов в общем случае избежать невозможно, однако она не так опасна для системы, как неограниченная. Для того чтобы избежать неограниченной инверсии приоритетов, используются два протокола изменения приоритетов задач:

Протокол наследования приоритета (Priority inheritance protocol)

Протокол увеличения приоритета (Priority ceiling protocol)

5.2.1. Протокол наследования приоритета

Допустим в системе существуют две задачи с низким (А) и высоким (В) приоритетом:



В момент Т2 задача (В) вытесняет низкоприоритетную задачу (А) и затем в момент времени Т3 пытается захватить заблокированный (А) ресурс.

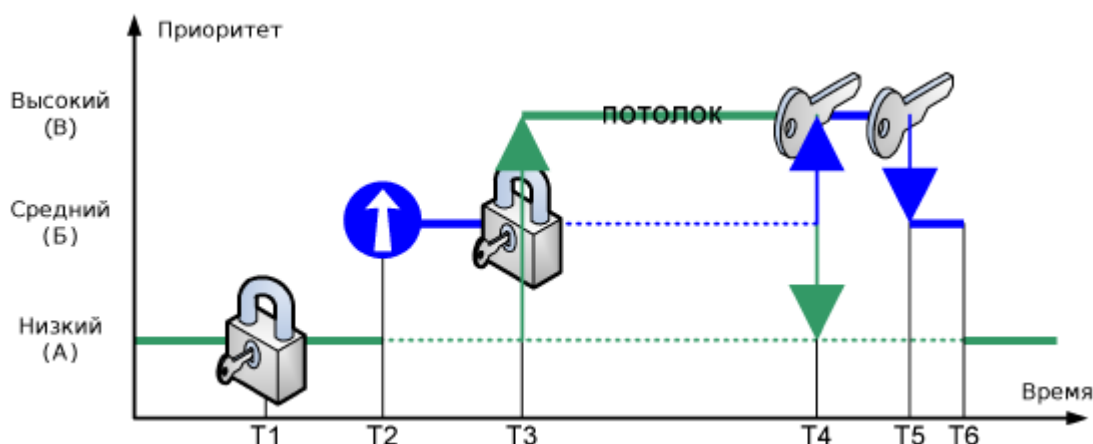
Протокол наследования приоритета состоит в том, что приоритет задачи (А) повышается до приоритета задачи (В) в момент времени Т3, то есть когда (В) пытается захватить заблокированный ресурс. Таким образом задачи с приоритетом больше (А) но меньше (В) не могут реализовать неограниченную инверсию, и задача (В) получит ресурс сразу после того как (А) его разблокирует.

После того как задача (А) разблокирует ресурс, ее приоритет понижается до исходного.

5.2.2. Протокол увеличения приоритета

Протокол увеличения приоритета основан на том факте, что на момент проектирования известны все задачи, которым требуется определенный ресурс, а так же известны приоритеты этих задач. В этом случае ресурсу (мютексу) можно назначить определенное свойство - максимальный приоритет из всех задач, которые могут его заблокировать (**потолок**).

Допустим в системе существуют три задачи с низким (А), средним (Б) и высоким (В) приоритетом, которые могут заблокировать один ресурс:

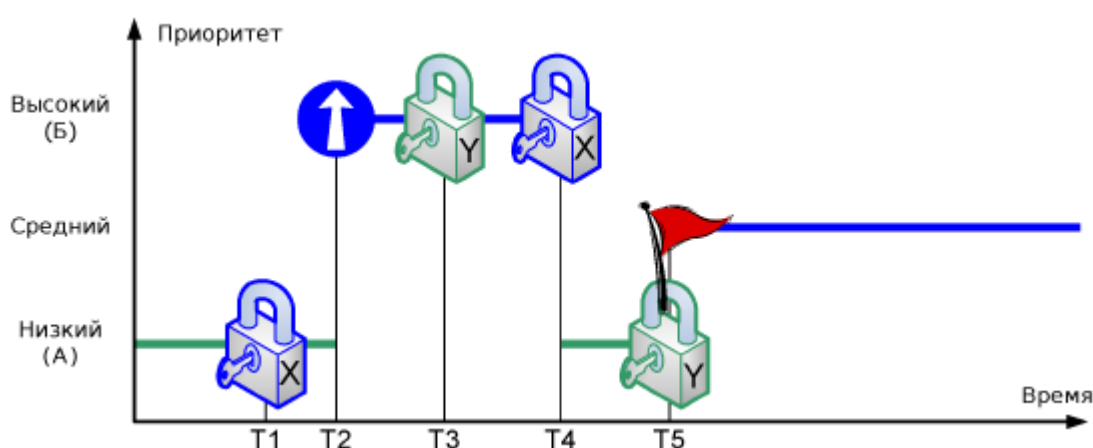


В момент времени T3, когда задача (Б) пытается захватить заблокированный (А) ресурс, приоритет (А) *повышается* до приоритета задачи (В), т.е. до максимального приоритета из всех задач, которые могут владеть ресурсом. Как только задача (А) освобождает ресурс в момент времени T4, ее приоритет понижается до исходного, а приоритет задачи (Б), ожидавшей ресурс повышается до (В).

Таким образом, при использовании протокола увеличения приоритета, задача, захватившая ресурс всегда имеет наивысший приоритет из группы задач, которые могут этим ресурсом владеть. Это позволяет не только избавиться от неограниченной инверсии приоритетов, но и не допустить взаимных блокировок.

5.3. Взаимная блокировка

Взаимная блокировка - это аварийное состояние системы, которое может возникать при вложенности блокировок ресурсов. Допустим в системе существуют две задачи с низким (А) и высоким (Б) приоритетом, которые используют два ресурса - X и Y:



В момент времени T1 задача (А) блокирует ресурс X. Затем в момент времени T2 задачу (А) вытесняет более приоритетная задача (Б), которая в момент времени T3 блокирует ресурс Y. Если задача (Б) попытается заблокировать ресурс X (T4) не освободив ресурс Y, то она будет переведена в состояние ожидания, а выполнение задачи (А) будет продолжено. Если в момент времени T5 задача (А) попытается заблокировать ресурс Y, не освободив X, возникнет состояние взаимной блокировки - ни одна из задач (А) и (Б) не сможет получить управление.

Взаимная блокировка возможна только если в системе используются вложенный конкурентный доступ к ресурсам. Взаимной блокировки можно избежать, если не использовать вложенность, или если ресурс использует протокол увеличения приоритета.

5.4. Структура управления мьютексом

В TNKernel реализованы мьютексы как с протоколом наследования приоритета, так и с протоколом увеличения приоритета.

Протокол наследования приоритета более простой и быстрый, но не позволяет избежать взаимной блокировки. Поэтому для таких мьютексов не рекомендуется использовать вложенный доступ. Протокол увеличения приоритета требует больше временных ресурсов и обеспечивает отсутствие взаимной блокировки.

Каждый мьютекс ассоциируется со структурой управления:

```
typedef struct _TN_MUTEX_S
{
    CDLL_QUEUE_S    wait_queue;
    CDLL_QUEUE_S    mutex_queue;
    CDLL_QUEUE_S    lock_mutex_queue;
    TN_UWORD        attr;

    TN_TCB_S        * holder;
    TN_UWORD        ceil_priority;
    TN_WORD         cnt;
    TN_OBJ_ID       id_mutex;
} TN_MUTEX_S;
```

В состав структуры мьютекса входят следующие элементы:

<code>wait_queue</code>	Очередь задач, ожидающих освобождение мьютекса
<code>mutex_queue</code>	Элемент списка заблокированных задач мьютексов
<code>lock_mutex_queue</code>	Системная очередь заблокированных мьютексов
<code>attr</code>	Атрибут (тип обхода инверсии приоритетов) мьютекса
<code>holder</code>	Указатель на TCB задачи, блокирующей мьютекс
<code>ceil_priority</code>	Максимальный приоритет из задач, которые могут использовать ресурс (требуется для протокола увеличения приоритета)
<code>cnt</code>	Зарезервировано
<code>id_mutex</code>	Поле идентификации объекта как мьютекса



Структура мьютекса доступна только при определении `TN_DEBUG`. Тем не менее, прямой доступ к элементам структуры мьютекса крайне не рекомендуется, так как это является вмешательством в работу планировщика и других сервисов RTOS.

5.5. Сервисы управления мютексами

TNKernel имеет следующий набор функций (сервисов) для управления мютексами:

Сервис	Описание	Свойства
--------	----------	----------

Создание и удаление мютекса

<u>tn_mutex_create()</u>	Создание мютекса
--	------------------



<u>tn_mutex_delete()</u>	Удаление мютекса
--	------------------



Блокировка мютекса

<u>tn_mutex_lock()</u>	Блокировка мютекса
--	--------------------



<u>tn_mutex_lock_polling()</u>	Попытка блокировки мютекса без блокировки задачи
--	--



Освобождение мютекса

<u>tn_mutex_unlock()</u>	Освобождение мютекса
--	----------------------



6. TNKernel : Блоки памяти фиксированного размера

6.1. Введение

Стандартные функции `malloc/free` как правило не являются безопасными с точки зрения многозадачности, поэтому необходимо либо отказаться от их применения, либо рассматривать кучу (`heap`) как разделяемый ресурс (использовать мьютекс), либо реализовать менеджер памяти своими силами под конкретную задачу.

Пул блоков памяти фиксированного размера - объект RTOS, предназначенный для динамического выделения памяти в многозадачной среде - может частично решить эту проблему. Пул представляет собой набор блоков памяти фиксированного размера (как правило, кратного машинному слову) и управляющую структуру, которая определяет занятые и свободные блоки.

Задача может получить блок памяти. Если в пуле нет свободных блоков, то задача переводится в состояние ожидания до тех пор, пока один из блоков не освободится или пока не истечет таймаут сервиса запроса. После использования блока задача может освободить его.

6.2. Структура управления пулом блоков памяти

Каждый пул ассоциируется со структурой управления:

```
typedef struct _TN_FMP_S
{
    CDLL_QUEUE_S    wait_queue;

    TN_UWORD        block_size;
    TN_UWORD        num_blocks;
    void            * start_addr;
    void            * free_list;
    TN_UWORD        fblkcnt;
    TN_OBJ_ID       id_fmp;
} TN_FMP_S;
```

В состав структуры пула входят следующие элементы:

`wait_queue` Очередь задач, ожидающих освобождение блока

`block_size` Размер блока памяти в байтах

`num_blocks` Количество блоков в пуле

`start_addr` Указатель на область памяти, выделенную для пула

`free_list` Указатель на список свободных блоков

`fblkcnt` Количество свободных блоков

`id_fmp` Поле идентификации объекта как пула блоков памяти



Структура пула доступна только при определении `TN_DEBUG`. Тем не менее, прямой доступ к элементам структуры пула блоков памяти крайне не рекомендуется, так как это является вмешательством в работу планировщика и других сервисов RTOS.

6.3. Сервисы управления пулами блоков памяти

TNKernel имеет следующий набор функций (сервисов) для управления пулами:

Сервис	Описание	Свойства
--------	----------	----------

Создание и удаление пула

[tn_fmem_create\(\)](#)

Создание пула



[tn_fmem_delete\(\)](#)

Удаление пула



Получение блока памяти

[tn_fmem_get\(\)](#)

Получение блока памяти



[tn_fmem_get_polling\(\)](#)

Получение блока памяти без блокировки задачи



[tn_fmem_get_ipolling\(\)](#)

Получение блока памяти в прерывании



Освобождение блока памяти

[tn_fmem_release\(\)](#)

Освобождение блока памяти



[tn_fmem_irelease\(\)](#)

Освобождение блока памяти в прерывании



7. TNKernel : Системные сервисы

7.1. Введение

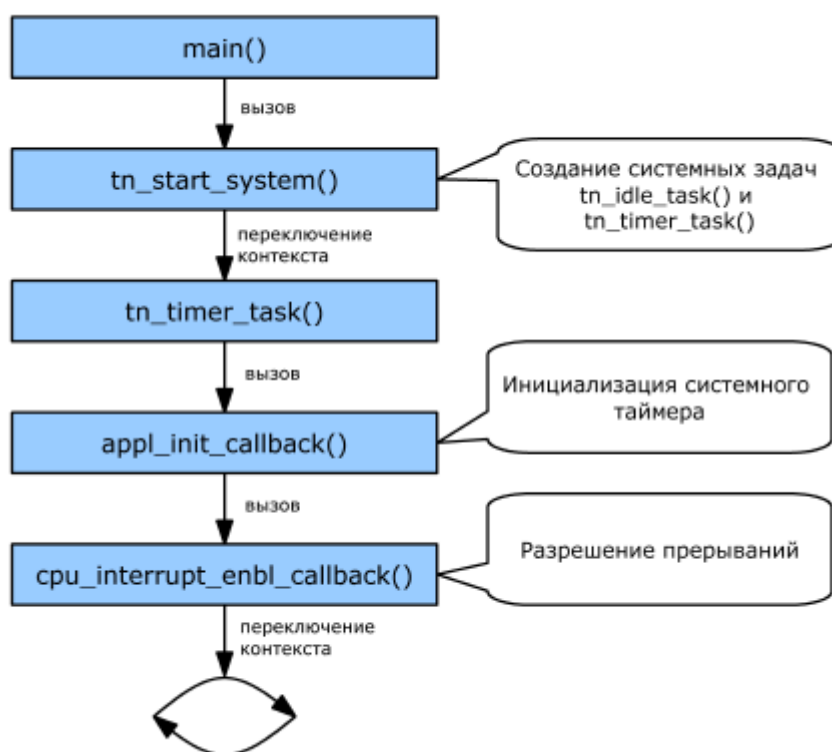
Системные сервисы не относятся напрямую к каким-либо объектам RTOS, они позволяют управлять системой в целом или получать текущие параметры системы. К системным сервисам относятся функция запуска `tn_start_system()`, функция управления карусельным планированием `tn_sys_tslice_ticks()`, функция обработчика системного таймера `tn_tick_int_processing()`, функции реализации критической секции `tn_sys_enter_critical()` и `tn_sys_exit_critical()`, функции обслуживания системного времени `tn_sys_time_set()` и `tn_sys_time_get()`.

7.2. Запуск системы

Функция запуска системы `tn_start_system()` обеспечивает инициализацию всех внутренних структур RTOS, создание двух [системных задач](#) и первое переключение контекста на системную задачу таймера. Функция `tn_start_system()` вызывается только один раз и является функцией без возврата.

В порте TNKernel для контроллеров PIC24/dsPIC в функцию передается несколько параметров, позволяющих более гибко настроить систему (выбрать размеры стеков системных задач и т.п.). В оригинальной версии функция запуска параметров не имеет.

Диаграмма запуска TNKernel изображена на рисунке:



Сервис запуска `tn_start_system()` как правило вызывается из функции `main()`.



Системные прерывания должны быть запрещены до момента вызова `tn_start_system()`.

Функция `tn_start_system()` создает две системные задачи `tn_idle_task()` и `tn_timer_task()` и запускает задачу `tn_timer_task()`, которая при старте последовательно вызывает две callback-

функции - `appl_init_callback()` и `cpu_interrupt_enbl_callback()`. Указатели на эти функции передаются в систему в вызове `tn_start_system()`.

Функция `appl_init_callback()` служит для инициализации системного таймера, периферийных модулей, создания необходимых задач. В функции `cpu_interrupt_enbl_callback()` разрешается прерывание от системного таймера и после выхода из нее система начинает нормальное функционирование.

7.3. Системный таймер

Для того чтобы реализовать функции ожидания или таймауты, в системе должен присутствовать таймер, доступный планировщику. Назовем такой таймер **системным таймером**. Как правило, период таймера постоянен на всем протяжении работы и это период называется **системным тиком**. Системный тик - это минимальная единица времени доступная планировщику. Все времена, таймауты указываются именно в системных тиках а не в абсолютных единицах времени.

Обычно системный таймер реализуется на основании одного из аппаратных таймеров микроконтроллера, или таймера, являющегося частью ядра. В обработчике прерывания от этого таймера необходимо вызвать функцию `tn_tick_int_processing()`, которая и обеспечивает "ход" системного времени.



Функция `tn_tick_int_processing()` должна вызываться только в обработчике прерывания.

7.4. Управление Round-Robin

Round-Robin или карусельное планирование - это принцип переключения задач с **одинаковым** приоритетом при котором каждой задаче выделяется определенный квант времени (с дискретностью один системный тик). После того как задача отработает свой квант, планировщик запускает следующую в очереди готовых к выполнению задачу.

В TNKernel включен сервис `tn_sys_tslice_ticks()`, позволяющий настраивать длительность кванта выполнения для каждого приоритета или отключать карусельное планирование.



Карусельное планирование отключено по умолчанию для всех приоритетов.

7.5. Запрещение переключения контекста

Критическая секция это часть задачи, в которой осуществляется доступ к разделяемому ресурсу. Если один и тот же ресурс (например, глобальную переменную) используют две или более задач, критические секции называют конкурирующими. В этом случае необходимо защитить критическую секцию таким образом, чтобы доступ к ресурсу являлся атомарным.

Один из способов защиты критической секции - это использование мютекса. Однако часто мютекс является избыточным объектом для реализации критической секции и в этом случае используют парные функции запрещения и разрешения переключения контекста.

В TNKernel для PIC24/dsPIC переключение контекста запрещает функция `tn_sys_enter_critical()` и разрешает функция `tn_sys_exit_critical()`.

Вызов функций `tn_sys_enter_critical()` и `tn_sys_exit_critical()` может быть несимметричным и вложенным. Например, допустимо следующее:

```
void foo(void)
{
    tn_sys_enter_critical();
    /* ... */
}

void TN_TASK task_1(void *param)
{
    for (;;)
    {
```

```

/* ... */

tn_sys_enter_critical();
foo();
tn_sys_exit_critical();

/* ... */
}
}

```

Однако рекомендуется использовать симметричный вызов функций запрещения и разрешения планирования, так как обратное может привести к логическим ошибкам.

Запрещение переключения контекста не приветствуется, это является вмешательством в работу планировщика. Однако функции `tn_sys_enter_critical()` и `tn_sys_exit_critical()` могут быть полезны для выполнения относительно быстрых операций, для которых использование мьютекса слишком избыточно. Примером такой операции может служить [вывод в порт](#) контроллера.

7.6. Системное время

Под системным временем подразумевается беззнаковая целая переменная, инкрементируемая каждый системный тик. Значение переменной может быть получено путем вызова функции `tn_sys_time_get()` и установлено с помощью вызова `tn_sys_time_set()`.

Системное время можно использовать для подсчета времени выполнения какого-либо действия (с точностью плюс-минус системный тик) или для "подстройки" периода "вызова" задач.

Допустим в системе присутствует задача, период "вызова" которой должен быть строго постоянным. Обычно такое поведение реализуется следующим образом.

```

void TN_TASK task_1 (void *param)
{
    for (;;)
    {
        foo();
        tn_task_sleep(FOO_PERIOD);
    }
}

```

Возможна ситуация, когда длительность выполнения функции `foo()` может превышать системный тик - в этом случае периодичность нарушается. Способов решения такой проблемы несколько - например, дополнительная задача, освобождающая семафор с фиксированным периодом или использование объекта типа "таймер". Однако первое ведет к дополнительной трате ресурсов, а второе в TNKernel пока не реализовано.

Используя системное время проблему можно решить следующим образом:

```

void TN_TASK task_1 (void *param)
{
    TN_SYS_TIM_T t;









    for (;;)
    {
        t = tn_sys_time_get();
        foo();
        t = tn_sys_time_get() - t;

        if (t < FOO_PERIOD)
            tn_task_sleep(FOO_PERIOD - t);
        else
            tn_task_sleep(1);
    }
}

```

7.7. Системные сервисы

TNKernel имеет следующий набор системных сервисов:

Сервис	Описание	Свойства
Основные сервисы		
tn_start_system()	Запуск системы	до начала работы системы
tn_tick_int_processing()	Обслуживание системного таймера	 
tn_sys_tslice_ticks()	Управление round-robin планированием	
tn_sys_context_get()	Получение текущего контекста системы	
Запрещение переключения контекста		
tn_sys_enter_critical()	Вход в критическую секцию	
tn_sys_exit_critical()	Выход из критической секции	
Системное время		
tn_sys_time_get()	Получение системного времени	
tn_sys_time_set()	Установка системного времени	

8. Отличия TNKernel для PIC24/dsPIC и PIC32

Исходные коды

Структура исходных текстов TNKernel значительно изменена по сравнению с оригинальной. Основное отличие - каждая функция находится в отдельном файле. Таким образом обходится проблема линкера C30, который не может игнорировать неиспользуемые секции кода.

При сборке проекта в исполняемый файл линкер добавляет только те функции, которые используются в пользовательском приложении. Это позволяет значительно сократить объем программной памяти, используемый ядром. Например, все сервисы требуют порядка 15 кБ программной памяти, тогда как в среднем приложении ядро RTOS занимает примерно 6-7 кБ.

Кроме разделения исходных кодов на файлы была предпринята попытка сделать TNKernel еще более портируемой - все основные типы переопределяются, все машинозависимые функции вынесены в отдельные модули (префикс `port_`). Это позволило в свое время достаточно просто добавить порт для PIC32.

Итак, скачав архив с проектом вы увидите каталог `source` в котором и находятся исходные тексты TNKernel для PIC24/dsPIC. Файл `_build_mchp_c30.bat` предназначен для сборки библиотеки. Его можно отредактировать для сборки файла с требуемыми параметрами - другим уровнем оптимизации, моделью памяти и пр. Текущая версия командного файла собирает четыре библиотеки - две для PIC24 с оптимизацией `Os` и форматом `coff` и `elf` и две для dsPIC. То же самое и для PIC32, командный файл называется `_build_mchp_c32.bat`

Для сборки необходимо, чтобы в системе был прописан путь к исполняемому файлу компилятора.

Порт для PIC24/dsPIC

Компания Microchip имеет две основные линейки 16-битных контроллеров: [PIC24](#) - микроконтроллеры общего назначения и [dsPIC](#) - контроллеры цифровой обработки сигналов. По сути PIC24 являются усеченной версией dsPIC - в них отсутствует DSP ядро и специальные методы адресации.

Изначально TNKernel портировалась как под PIC24, так и под dsPIC, причем в версии для последнего в стеке задачи кроме всего прочего, сохранялся контекст DSP-ядра. Но как выяснилось это не имело большого смысла, потому, что полностью восстановить контекст DSP простыми способами невозможно - большинство статусных флагов DSP-ядра имеют доступ только для чтения, да и с аппаратным циклом DO все не так просто. Поэтому от отдельного порта для dsPIC было решено отказаться - в следующих вариантах TNKernel для PIC24/dsPIC DSP-ядро было предложено рассматривать как разделяемый ресурс и использовать для доступа к нему из разных задач мьютекс.

Но как выяснилось напрасно. Благодаря камраду **qas** был найден серьезный [баг](#), который мог однозначно порушить систему при использовании модульной или бит-реверсивной адресации DSP-ядра. Баг исправлен, но опять появилось две версии TNKernel - для PIC24 и для dsPIC.

Тем не менее ситуация с DSP-ядром прежняя - его контекст не сохраняется и его следует рассматривать как разделяемый ресурс, используя для доступа из разных задач мьютекс.

Порт для PIC32

Порт для PIC32 был реализован весной 2010 года по просьбам трудящихся. По сути он мало чем отличается от порта для PIC24/dsPIC. Начиная с версии 2.5.600 в комплекте идет пример, который может быть скомпилирован как под PIC24 так и под PIC32 без изменений.

8.1. Основные отличия от оригинальной версии

8.1.1. Типы данных

Все стандартные типы данных (кроме `void`) переопределены:

TN_CHAR

для всех архитектур соответствует `signed char`. Под `char` в данном случае подразумевается 1 байт.

TN_UCHAR

для всех архитектур соответствует `unsigned char`

TN_WORD

для PIC24/dsPIC (компилятор C30) соответствует `signed int`, то есть 16-битному целому со знаком

для ARM (Keil RV) и PIC32 (C32) соответствует `signed int`, то есть 32-битному целому со знаком

TN_UWORD

размер машинного слова. Этот тип рекомендуется использовать для объявления стеков задач

для PIC24/dsPIC (компилятор C30) соответствует `unsigned int`, то есть беззнаковому 16-битному целому

для ARM (Keil RV) и PIC32 (C32) соответствует `unsigned int`, то есть беззнаковому 32-битному целому

TN_SYS_TIM_T

тип счетчика [системного времени](#). Для PIC24/dsPIC это 32-битный счетчик, для ARM/PIC32 - 64-битный

TN_TIMEOUT

тип таймута

для PIC24/dsPIC (компилятор C30) соответствует `unsigned int`, то есть беззнаковому 16-битному целому

для ARM (Keil RV) и PIC32 (C32) соответствует `unsigned int`, то есть беззнаковому 32-битному целому

Рекомендуется употреблять эти типы для объявления переменных и массивов, связанных непосредственно с системой - стеков задач, блоков памяти фиксированного размера, очередей сообщений и др.:

TN_UWORD task_1_stack[128] TN_DATA;

8.1.2. Приоритеты задач

В оригинальной версии TNKernel задачи могут иметь приоритет от **1** до **30** (**0** и **31** приоритет имеют системные задачи). В версии TNKernel для PIC24/dsPIC пользовательские задачи могут иметь приоритет от **1** до **14** (**0** и **15** приоритет имеют системные задачи). Это связано с разрядностью слова контроллера и стремлением сократить время поиска следующей задачи, т.е. по сути время переключения контекста.

Следует сказать, что такого количества приоритетов вполне достаточно, так как опционально TNKernel обеспечивает карусельное (round-robin) переключение между задачами с одинаковым приоритетом.



В версии TNKernel для PIC32 задачи могут иметь такие же приоритеты как и в оригинальной: 1 до 30.

8.1.3. Инициализация системы

Инициализация системы в оригинальной версии TNKernel выполняется с помощью функции `tn_start_system()`, которая не имеет параметров. В порте TNKernel для PIC24/dsPIC и PIC32 эта функция выглядит следующим образом:

Вызов:

```
void tn_start_system (TN_UWORD *timer_task_stack,
                     TN_UWORD timer_task_stack_size,
                     TN_UWORD *idle_task_stack,
                     TN_UWORD idle_task_stack_size,
                     void (*app_in_cb)(void),
                     void (*cpu_int_en)(void),
                     void (*idle_user_cb)(void)
                     );
```

Разрешен вызов:

В контексте задачи

Параметры функции:

`timer_task_stack`

указатель на стек системной задачи таймера

`timer_task_stack_size`

размер стека системной задачи таймера (в машинных словах)

`idle_task_stack`

указатель на стек системной задачи простоя

`idle_task_stack_size`

размер стека системной задачи простоя (в машинных словах)

`app_in_cb`

указатель на функцию инициализации приложения. Эта функция вызывается после того как системные задачи будут созданы, а планировщик запущен

`cpu_int_en`

указатель на функцию конфигурации прерываний. Эта функция вызывается сразу после функции `app_in_cb`

`idle_user_cb`

указатель на функцию, циклически вызываемую из задачи простоя. В этой функции можно, например, инкрементировать счетчик загрузки или уводить контроллер в состояние пониженного энергопотребления (Sleep или Idle)

Возвращаемые значения:

нет

Пример вызова:

```
#define TMR_TASK_STACK_SIZE 128
#define IDL_TASK_STACK_SIZE 128

TN_UWORD stk_tmr[TMR_TASK_STACK_SIZE] TN_DATA; /* стек задачи таймера */
TN_UWORD stk_idl[IDL_TASK_STACK_SIZE] TN_DATA; /* стек задачи простоя */

void appl_init(void);
void intr_init(void);
void idle_user(void);

int main (void)
{
    tn_start_system(stk_tmr,
                    TMR_TASK_STACK_SIZE,
                    stk_idl,
                    IDL_TASK_STACK_SIZE,
                    appl_init,
                    intr_init,
                    idle_user
                    );
}

void appl_init (void)
{
    /* инициализация */
}

void intr_init (void)
{
    /* инициализация и разрешение прерываний */
}

void idle_user (void)
{
    /* когда нечего делать мы тут */
}
```

Функции `app_in_cb` и `cpu_int_en` заменяют `tn_app_init()` и `tn_cpu_int_enable()` в оригинальной версии `TNKernel`.

Введение параметров в функцию `tn_start_system()` позволило более гибко настраивать систему, в частности, выбирать размеры стеков системных задач и выполнять полезные действия в задаче простоя (`tn_task_idle()`).

По сути в точке входа приложения - функции `main()` должен вызываться только сервис `tn_start_system()`.

8.1.4. Создание задачи

Изменен вызов сервиса создания задачи `tn_task_create()`:

```
TN_RETVAL tn_task_create (TN_TCB *task,
                          void (*task_func)(void *param),
                          TN_UWORD priority,
                          TN_UWORD *task_stack_start,
                          TN_UWORD task_stack_size,
                          void *param,
                          TN_UWORD option
                          );
```

Параметр `task_stack_start` указывает на вершину (младший адрес) стека задачи, тогда как в оригинальной версии, `task_stack_start` указывает на старший адрес стека. Это связано с тем, что в PIC24/dsPIC стек растет от младшего адреса к старшему.



В версии TNKernel для PIC32 в функцию создания задачи так же должен передаваться адрес вершины стека, несмотря на то, что у MIPS32 стек растет от старшего адреса к младшему.

8.2. Нововведения

8.2.1. 1. Критические секции

Добавлены функции `tn_sys_enter_critical()` и `tn_sys_exit_critical()`, которые аналогичны используемым в оригинальной версии `tn_disable_interrupt()` и `tn_enable_interrupt()`. Функции используются следующим образом:

```
/* ... */

tn_sys_enter_critical();
/*
 критическая секция кода, в которой запрещено переключение контекста
 */
tn_sys_exit_critical();

/* ... */
```

Названия функций отражают их назначение - выделение части кода в критическую секцию в которой запрещено переключение контекста. `tn_disable_interrupt()` и `tn_enable_interrupt()` - не совсем корректное название для PIC24/dsPIC, которые имеют векторный приоритетный контроллер прерываний.



В версии TNKernel PIC32 функция `tn_sys_enter_critical()` запрещает все прерывания!

8.2.2. Новые сервисы

Добавлены следующие сервисы:

[`tn_task_isuspend\(\)`](#) - останов задачи в прерывании

[`tn_task_iresume\(\)`](#) - восстановление задачи из прерывания

[`tn_sys_context_get\(\)`](#) - получение текущего контекста системы

[`tn_task_reference\(\)`](#) - получение информации о задаче

[`tn_task_ireference\(\)`](#) - получение информации о задаче в прерывании

8.2.3. Атрибут задачи

Функции задач могут объявляться с атрибутом `TN_TASK`. Этот атрибут сообщает компилятору о том, что функция имеет бесконечный цикл и выхода из нее не будет. В большинстве случаев это позволяет уменьшить размер стека задачи. Пример:

```
void TN_TASK Task (void *par)
{
    for (;;)
    {
        tn_task_sleep(10);
    }
}
```

8.2.4. Атрибут данных

Объекты и стеки задач могут объявляться с атрибутом `TN_DATA`. По сути он размещает переменные в отдельной секции ОЗУ - это позволяет контролировать объем памяти, занимаемой объектами RTOS и стеками задач. Для этого в скрипт линкера необходимо добавить следующие строки (см., например, файл `..\example1\p24FJ128GA006.gld`):

```
.tnk_data :  
{  
    *(tnk_data);  
}> data
```

Пример использования атрибута:

```
TN_SEM Sem_From_IRQ TN_DATA;  
TN_DQUE que_test TN_DATA;
```

Все сервисы `TNKernel` размещаются в отдельную секцию кода. Это позволяет контролировать объем программной памяти, которую занимает ядро. Для этого в скрипт линкера необходимо добавить следующие строки (см., например, файл `..\example1\p24FJ128GA006.gld`):

```
.tnk_code :  
{  
    *(tnk_code);  
}>program
```



| В версии `TNKernel` для PIC32 именованные секции кода пока не поддерживаются.

8.2.5. Отладка

Если в заголовочном файле `tnkernel_conf.h` не объявить `TN_DEBUG`, внутренняя структура всех объектов будет скрыта от пользователя, и структура объектов в окне `Watch` отладчика будет отображена в виде байтового массива.

Если `TN_DEBUG` будет объявлен, структуры объектов будут раскрыты. Это позволит отлаживать приложение контролируя значения полей структур.

8.2.6. Варианты сервисов без проверки параметров

В порте `TNKernel` для PIC24/dsPIC и PIC32 имеется два набора сервисов - с проверкой параметров и без проверки параметров. Естественно, последние будут занимать меньше программной памяти и будут быстрее выполняться.

Объявление `TN_NO_ERROR_CHECKING` в файле конфигурации системы `tnkernel_conf.h` позволяет использовать более компактные и быстрые варианты сервисов без проверки параметров.

8.2.7. Контроль переполнения стеков задач

Микроконтроллеры PIC24/dsPIC имеют аппаратный механизм контроля переполнения стека, который полностью задействован в `TNKernel` для PIC24/dsPIC. Для того чтобы контролировать переполнение, необходимо объявить в коде исключение (trap) по ошибке стека:

```
void __attribute__((interrupt, no_auto_psv)) _StackError (void)  
{  
    for (;;) /* при переполнении стека задачи попадем сюда */  
}
```



| В версии `TNKernel` для PIC32 контроль переполнения стека не поддерживается

8.2.8. Код возврата TERR_EXS

Любой сервис, создающий объект (`tn_task_create()`, `tn_sem_create()`, `tn_queue_create()`, `tn_event_create()`, `tn_fmem_create()` и `tn_mutex_create()`), проверяет состояние объекта (уже создан или нет) и, либо продолжает работу, либо (если объект уже создан) возвращает код ошибки `TERR_EXS`.

Наличие проверки состояния объекта не зависит от типа вызываемого сервиса (с проверкой или без проверки параметров).

8.2.9. Получение ревизии TNKernel

Добавлен заголовочный файл `tnkernel_rev.h`, в котором присутствуют следующие определения:

`__TNKERNEL_VERSION` - текущая версия (float)

`__TNKERNEL_REVISION` - текущая ревизия (беззнаковое целое)

`__TNKERNEL_REVISION_TIME_STRING` - время и дата создания ревизии (строка)

`__TNKERNEL_BUILD_TIME_STRING` - время и дата сборки библиотеки (строка)

Пример использования:

```
TN_UWORD tn_revision = __TNKERNEL_REVISION;
char *tn_data = __TNKERNEL_REVISION_TIME_STRING;
char *tn_build = __TNKERNEL_BUILD_TIME_STRING;

#if (__TNKERNEL_VERSION == 2.5)
    #if (__TNKERNEL_REVISION == 977)
        /* ... */
    #endif
#else
    /* ... */
#endif

printf(tn_data);
```

8.2.10. Системный таймер

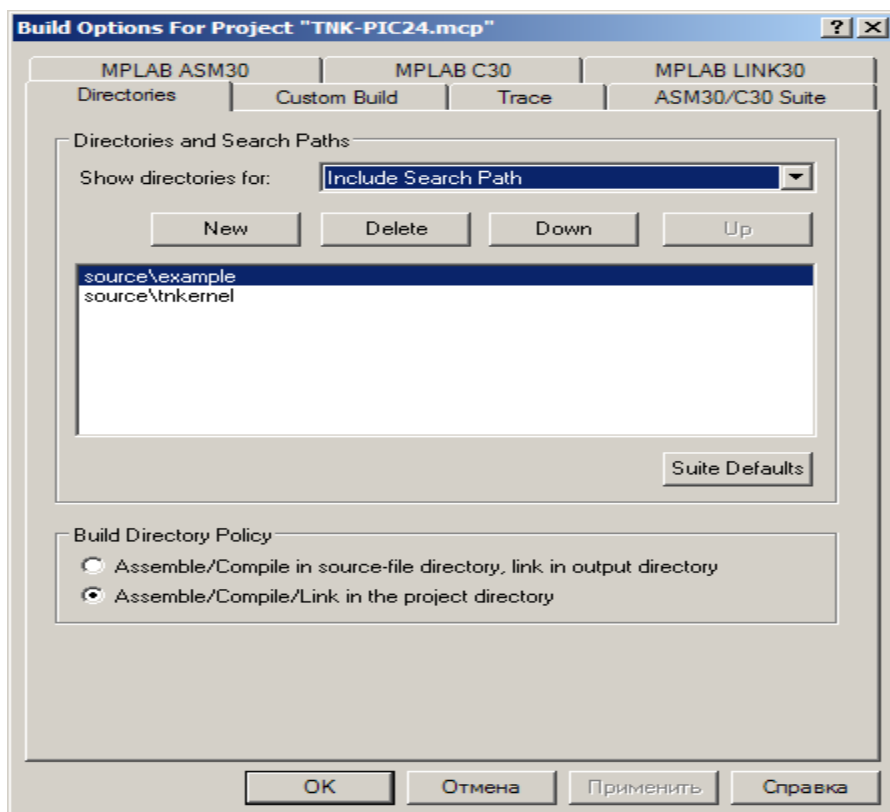
Системным таймером назван счетчик, инкрементируемый каждый системный тик. Для установки и получения значения системного таймера используются следующие сервисы:

[`tn_sys_time_set\(\)`](#) - установка системного таймера

[`tn_sys_time_get\(\)`](#) - получение значения системного таймера

8.2.11. Файл конфигурации

Пользовательский проект должен включать в себя заголовочный файл конфигурации `tnkernel_conf.h`, в котором определены (или не определены) дефайны `TN_DEBUG` и `TN_NO_ERROR_CHECKING`. Конечно, в свойствах проекта папка с этим файлом должна быть добавлена в пути поиска заголовочных файлов.



8.3. Использование прерываний

Основное предупреждение: все прерывания должны быть запрещены до момента запуска системы. Для конфигурации источников прерываний и разрешения прерываний предназначена функция `cpu_int_en`, указатель на которую передается в сервисе `tn_start_system()`.

Будем называть прерывания в которых вызываются сервисы системными, а все остальные прерывания - пользовательскими.



*В TNKernel для PIC24/dsPIC системные прерывания должны иметь приоритет, равный `TN_INTERRUPT_LEVEL` (приоритет 1). Вызов сервисов RTOS в обработчике прерывания с другим (более высоким) приоритетом (т.е. в ISR пользовательского прерывания) **запрещен** - это приведет к краху системы. В текущей версии TNKernel защита от вызова системных сервисов в пользовательском прерывании не реализована.*

*В TNKernel для PIC32 системные прерывания должны иметь **одинаковый** приоритет, но не обязательно равный `TN_INTERRUPT_LEVEL`. Однако для совместимости кода рекомендуется использовать приоритет 1, как и в версии TNKernel для PIC24/dsPIC.*

В TNKernel для PIC24/dsPIC и PIC32 не реализована вложенность системных прерываний. С одной стороны это может привести к задержке обработки прерывания, с другой - экономит стек задачи, что на самом деле более важно, особенно для PIC32. Если задержка входа в прерывание недопустима, можно использовать пользовательское прерывание с приоритетом большим чем

`TN_INTERRUPT_LEVEL`. Однако, не нужно забывать, что вызов сервисов RTOS в пользовательском прерывании запрещен, поэтому задачи должны взаимодействовать с пользовательским прерыванием с помощью глобальных переменных. Это некрасиво и по большому счету неправильно, но другого выхода нет...

Системные прерывания объявляются с помощью макроса `tn_sys_interrupt`, аргументом которого является зарезервированный псевдоним вектора прерывания:

```
/* PIC24/dsPIC */
tn_sys_interrupt(_INT0Interrupt) /* системное прерывание, источник INT0 */
{
    IFS0bits.INT0IF = 0;
    tn_queue_isend_polling(&que_test, transceived_buff);
}

/* PIC32 */
tn_sys_interrupt(_CHANGE_NOTICE_VECTOR)
{
    INTClearFlag(INT_CN);
    /* обработка прерывания */
}
```

Пользовательские прерывания объявляются обычным для C30/C32 способом.

Одно из системных прерываний всегда должно быть зарезервировано для системного таймера. Как правило это прерывание от аппаратного таймера с периодом 1-10 мс:

```

/* PIC24/dsPIC */
tn_sys_interrupt (_T2Interrupt) /* системное прерывание, источник TMR2 */
{
    IFS0bits.T2IF = 0;
    tn_tick_int_processing();
}

/* PIC32 */
tn_sys_interrupt(_CORE_TIMER_VECTOR) /* системное прерывание, источник - системный таймер MIPS32 */
{
    Sys_Tmr_Int_Handler();
    tn_tick_int_processing();
}

```



Сервис `tn_tick_int_processing()` должен вызываться только из системного прерывания.



***Внимание!!!** Если в обработке прерывания вызывается сервис `tn_tick_int_processing()`, то вызов других сервисов RTOS в этом прерывании запрещен!*

Следует заметить что сервисы `tn_sys_enter_critical()` и `tn_sys_exit_critical()` запрещают **системные** прерывания, в то время как прерывания с приоритетом, большим `TN_INTERRUPT_LEVEL` остаются активными - только для PIC24/dsPIC. Для PIC32 эти сервисы запрещают **ВСЕ** прерывания.

8.4. Отличия порта для PIC32 от порта для PIC24/dsPIC

`tn_sys_enter_critical()` запрещает все прерывания, а не только с приоритетом `TN_INTERRUPT_LEVEL`

данные и код не размещаются в именованные секции

системные прерывания могут иметь приоритет отличный от `TN_INTERRUPT_LEVEL`, однако приоритет должен быть одинаковым для всех системных прерываний

приоритеты задач - от 1 до 30

переполнение стека не контролируется!

9. Сервисы RTOS

9.1. Сервисы управления задачами

9.1.1. Создание и удаление задачи

tn_task_create()

Функция предназначена для создания задачи. Поле `id_task` TCB задачи `task` должно быть равно нулю до момента создания задачи, таким образом уже созданные задачи защищаются от повторного создания. Память для управляющей структуры TCB `task` и для стека задачи должна быть выделена до момента создания задачи. Память может быть выделена на этапе компиляции (объявление глобальной переменной типа `TN_TASK` для TCB задачи и массива с элементами типа `TN_UWORD` для стека задачи), либо динамически, если пользовательское приложение использует менеджер памяти.

Размер стека задачи `tn_task_size` должен быть выбран исходя из количества локальных переменных в функции задачи, дерева вызовов, количество и вложенности прерываний и других специфичных для конкретного приложения параметров. В любом случае стек должен полностью вмещать контекст задачи. В любом случае размер массива должен быть больше или равен `TN_MIN_STACK_SIZE`.

Стек задачи - это массив элементов типа `TN_UWORD`, разрядность `TN_UWORD` соответствует разрядности машинного слова микроконтроллера. Для ARM7 `sizeof(TN_UWORD) = 4`, для PIC24/dsPIC `sizeof(TN_UWORD) = 2`.

Параметр `task_stack_start` является указателем на вершину стека. Если в используемой архитектуре стек растёт от младшего адреса к старшему (PIC24/dsPIC), то параметр `task_stack_start` должен быть равен адресу первого элемента массива. В противном случае параметр должен быть равен последнему элементу массива.

Вызов:

```
TN_RETVAL tn_task_create(TN_TCB *task,
                        void (*task_func)(void *param),
                        TN_UWORD priority,
                        TN_UWORD *task_stack_start,
                        TN_UWORD task_stack_size,
                        void *param,
                        TN_UWORD option
                        );
```

Разрешен вызов: В контексте задачи

Параметры функции:

`task` указатель TCB задачи. Структура TCB должна быть создана до момента вызова функции, статически или динамически

`(*task_func)(void *param)` указатель на функцию задачи. Функция задачи имеет следующий прототип: `void (*task_func)(void *param)`

`priority` приоритет задачи. Пользовательские задачи могут иметь приоритет от 1 до `TN_NUM_PRIORITY - 1` включительно (приоритеты 0 и `TN_NUM_PRIORITY` зарезервированы для служебных задач)

`task_stack_start` указатель на стек задачи. Для PIC24/dsPIC - указатель на первый элемент массива стека задачи

`task_stack_size` размер стека задачи в машинных словах (количество элементов в массиве стека)

`param` арамтр передаваемый в функцию задачи

`option` параметр создания задачи, может принимать одно из двух значений:

TN_TASK_DORMANT_ON_CREATION после создания задача переводится в состояние *DORMANT*

TASK_START_ON_CREATION после создания задача переводится в состояние *RUNNABLE*

Возвращаемые значения:

TERR_WRONG_PARAM некорректное значение параметра (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

TERR_WCONTEXT попытка создания задачи в прерывании или в пользовательской критической секции

TERR_EXS попытка создания задачи, которая уже создана

TERR_NO_ERR успешное выполнение

Пример вызова:

```
#define TASK_1_STACK_SIZE    128            /* размер стека задачи */
#define TASK_1_PRIORITY      11            /* приоритет задачи */

TN_TCB   tcb_Task1 TN_DATA;               /* TCB задачи */
TN_UWORD stk_Task1[TASK_1_STACK_SIZE] TN_DATA; /* стек задачи */

void TN_TASK Task1(void *par);            /* прототип функции задачи */

/* ... создание задачу... */

    tn_task_create(&tcb_Task1,
                  Task1,
                  TASK_1_PRIORITY,
                  stk_Task1,
                  TASK_1_STACK_SIZE,
                  TN_NULL,
                  TN_TASK_START_ON_CREATION
                  );

/* ... */

void TN_TASK Task1 (void *par)            /* функция задачи */
{
    for (;;)
    {
        tn_task_sleep(10);
    }
}
```

tn_task_delete()

Функция удаляет задачу, находящуюся в состоянии *DORMANT*. Если удаляемая задача будет находиться в другом состоянии, сервис вернет код ошибки.

Функция сбрасывает поле *id_task* TCB задачи, и удаляет задачу из списка доступных в системе. Освободившуюся память можно использовать для создания другой задачи. Процесс удаления необратимый - для запуска удаленной задачи нужно создать ее заново функцией *tn_task_create*.

Вызов:

```
TN_RETVAL tn_task_delete(TN_TCB *task);
```

Разрешен вызов: В контексте задачи

Параметры функции:

task указатель на TCB удаляемой задачи

Возвращаемые значения:

TERR_WRONG_PARAM	некорректное значение параметра (<i>замечание:</i> данный код возврата возможен только в случае использования сервисов с проверкой параметров)
TERR_NOEXS	попытка удаления объекта, не являющегося задачей (<i>замечание:</i> данный код возврата возможен только в случае использования сервисов с проверкой параметров)
TERR_WCONTEXT	попытка удаления задачи в прерывании или в пользовательской критической секции удаляемая задача находится в состоянии, отличном от <i>DORMANT</i>
TERR_NO_ERR	успешное выполнение

Пример вызова:

```
extern TN_TCB tcb_Task1 TN_DATA;  
  
/* ... */  
if (need_delete_task_1)  
    tn_task_delete(&tcb_Task1);  
/* ... */
```

9.1.2. Перезапуск задачи

`tn_task_exit()`

Функция прекращает выполнение текущей задачи, при этом задача перемещается в состояние *DORMANT*. Все ресурсы ([мютексы](#)) занятые задачей разблокируются. Задача может быть дополнительно удалена, если сервис вызван с параметром `TN_EXIT_AND_DELETE_TASK`.

Если счетчик запросов на активацию задачи больше 1, то счетчик декрементируется, а задача переводится в состояние *READY*. Как только задача станет наиболее приоритетной, она запустится с точки входа в функцию задачи (так же как и первый раз после создания). Следует заметить, что задача будет поставлена в очередь готовых к выполнению (этого же приоритета) последней.

Задача может быть выведена из состояния *DORMANT* с помощью сервисов `tn_task_activate()` и `tn_task_iactivate()`. В этом случае задача будет запущена с точки входа в функцию задачи и поставлена в очередь готовых к выполнению (этого же приоритета) последней.

Функция `tn_task_exit()` может использоваться только для прекращения выполнения текущей задачи. Для прекращения выполнения другого потока необходимо использовать функцию `tn_task_terminate()`.

Вызов:

```
void tn_task_exit (TN_UWORD attr);
```

Разрешен вызов: В контексте задачи

Параметры функции:

`Attr` параметр, который указывает, будет ли задача удалена после прекращения выполнения:

<code>TN_EXIT_TASK</code>	функция прекратит выполнение текущей задачи. Для запуска задачи необходимо использовать сервисы <code>tn_task_activate()</code> или <code>tn_task_iactivate()</code>
<code>TN_EXIT_AND_DELETE_TASK</code>	после прекращения выполнения задача будет удалена. Для запуска удаленной задачи нужно создать ее заново функцией <code>tn_task_create</code>

Возвращаемые значения: нет

Пример вызова:

```
extern TN_TCB tcb_Task1 TN_DATA;

/* ... */
void Task1 (void *param)
{
    for (;;)
    {
        if (need_restart_task_1)
            tn_task_exit(TN_EXIT_TASK);
        /* ... */
    }
}
```

tn_task_terminate()

Функция переводит задачу из любого состояния в состояние *DORMANT*. Если задача находится в очереди ожидания какого-либо события или объекта, она удаляется из очереди, а все ресурсы (мютексы) используемые задачей будут освобождены.

Если счетчик запросов на активацию задачи больше 1, то счетчик декрементируется, а задача переводится в состояние *READY*. Как только задача станет наиболее приоритетной, она запустится с точки входа в функцию задачи (так же как и первый раз после создания). Следует заметить, что задача будет поставлена в очередь готовых к выполнению (этого же приоритета) последней.

Задача может быть выведена из состояния *DORMANT* с помощью сервисов *tn_task_activate()* и *tn_task_iactivate()*. В этом случае задача будет запущена с точки входа в функцию задачи и поставлена в очередь готовых к выполнению (этого же приоритета) последней.

Задача не может остановить свое выполнение с помощью сервиса *tn_task_terminate()*, для этого необходимо использовать сервис *tn_task_exit()*.

Вызов:

```
TN_RETVAL tn_task_terminate(TN_TCB *task);
```

Разрешен вызов: В контексте задачи

Параметры функции:

task указатель на TCB перезапускаемой задачи

Возвращаемые значения:

TERR_WRONG_PARAM	некорректное значение параметра (<i>замечание:</i> данный код возврата возможен только в случае использования сервисов с проверкой параметров)
TERR_NOEXS	попытка перезапуска объекта, не являющегося задачей (<i>замечание:</i> данный код возврата возможен только в случае использования сервисов с проверкой параметров)
TERR_WCONTEXT	попытка активации задачи в прерывании или в пользовательской критической секции : попытка перезапуска задачи, находящейся в состоянии <i>DORMANT</i> удаляемая задача находится в состоянии, отличном от <i>DORMANT</i>
TERR_NO_ERR	успешное выполнение

Пример вызова:

```
extern TN_TCB tcb_Task1 TN_DATA;  
  
/* ... */  
if (need_stop_task_1)  
    tn_task_terminate(&tcb_Task1);  
/* ... */
```

tn_task_activate()

Функция активизирует задачу - переводит задачу `task` из состояния *DORMANT* в состояние *READY*.

Вызов:

```
TN_RETVAL tn_task_activate(TN_TCB *task);
```

Разрешен вызов: В контексте задачи

Параметры функции:

`task` указатель на TCB активизируемой задачи

Возвращаемые значения:

`TERR_WRONG_PARAM` некорректное значение параметра (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

`TERR_NOEXS` попытка активации объекта, не являющегося задачей (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

`TERR_WCONTEXT` попытка активации задачи в прерывании или в пользовательской критической секции

`TERR_OVERFLOW` активируемая задача находится в состоянии, отличном от *DORMANT*

`TERR_NO_ERR` успешное выполнение

Пример вызова:

```
extern TN_TCB tcb_Task1 TN_DATA;

/* ... */
if (need_activate_task_1)
    tn_task_activate(&tcb_Task1);
/* ... */
```

tn_task_iactivate()

Функция активизирует задачу из прерывания - переводит задачу `task` из состояния *DORMANT* в состояние *READY*.

Вызов:

```
TN_RETVAL tn_task_iactivate(TN_TCB *task);
```

Разрешен вызов: В прерывании

Параметры функции:

`task` указатель на TCB активизируемой задачи

Возвращаемые значения:

`TERR_WRONG_PARAM` некорректное значение параметра (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

`TERR_NOEXS` попытка активации объекта, не являющегося задачей (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

`TERR_WCONTEXT` попытка активации задачи в контексте задачи или в пользовательской критической секции

`TERR_OVERFLOW` активируемая задача находится в состоянии, отличном от *DORMANT*

`TERR_NO_ERR` успешное выполнение

Пример вызова:

```
extern TN_TCB tcb_Task1 TN_DATA;

tn_sys_interrupt (_T3Interrupt)    /* прерывание от таймера */
{
    IFS0bits.T2IF = 0;             /* сброс флага прерывания */
    tn_task_iactivate(&tcb_Task1); /* активация задачи tcb_Task1 */
}
```

9.1.3. Останов и восстановление задачи

tn_task_suspend()

Функция приостанавливает выполнение задачи, переводя ее в состояние *SUSPENDED*. Если задача находится в состоянии *WAITING*, она переводится в состояние *WAITING_SUSPENDED*.

Функция может приостановить как выполнение текущей задачи, так и выполнение любой другой задачи.

Вызов:

```
TN_RETURN tn_task_suspend (TN_TCB *task);
```

Разрешен вызов: В контексте задачи

Параметры функции:

task указатель на TCB приостанавливаемой задачи

Возвращаемые значения:

TERR_WRONG_PARAM некорректное значение параметра (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

TERR_NOEXS попытка приостановки выполнения объекта, не являющегося задачей (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

TERR_WCONTEXT попытка приостановки выполнения задачи в прерывании или в пользовательской критической секции

TERR_OVERFLOW попытка приостановки выполнения задачи, которая уже находится в состоянии *SUSPENDED*

TERR_WSTATE попытка приостановки выполнения задачи, которая находится в состоянии *DORMANT*

TERR_NO_ERR успешное выполнение

Пример вызова:

```
extern TN_TCB tcb_Task1 TN_DATA;

/* ... */
if (need_suspend_task_1)
    tn_task_suspend(&tcb_Task1);
/* ... */
```

tn_task_isuspend()

Функция приостанавливает выполнение задачи, переводя ее в состояние *SUSPENDED* из прерывания. Если задача находится в состоянии *WAITING*, она переводится в состояние *WAITING_SUSPENDED*.

Функция может приостановить задачи выполняемой на момент получения запроса на прерывание, так и выполнение любой другой задачи.

Вызов:

```
TN_RETVAL tn_task_isuspend (TN_TCB *task);
```

Разрешен вызов: В прерывании

Параметры функции:

task указатель на TCB приостанавливаемой задачи

Возвращаемые значения:

*TERR_WRONG_PARAM*¹⁾ некорректное значение параметра

*TERR_NOEXS*²⁾ попытка приостановки выполнения объекта, не являющегося задачей

TERR_WCONTEXT попытка активации задачи в контексте задачи или в пользовательской критической секции

TERR_OVERFLOW попытка приостановки выполнения задачи, которая уже находится в состоянии *SUSPENDED*

TERR_WSTATE попытка приостановки выполнения задачи, которая находится в состоянии *DORMANT*

TERR_NO_ERR

успешное выполнение

Пример вызова:

```
extern TN_TCB tcb_Task1 TN_DATA;

tn_sys_interrupt (_T2Interrupt)    /* прерывание от таймера */
{
    IFS0bits.T2IF = 0;             /* сброс флага прерывания */
    tn_task_isuspend(&tcb_Task1);  /* приостановка выполнения задачи tcb_Task1 */
}
```


Функция предназначена для вывода задачи из состояния *SUSPENDED*.

Если задача находится в состоянии *SUSPENDED*, она переводится в состояние *READY*, при этом она будет поставлена в очередь готовых к выполнению последней. Если задача находится в состоянии *WAITING_SUSPENDED*, она будет переведена в состояние *WAITING* и продолжит ожидание события.

Вызов:

```
TN_RETVAL tn_task_resume (TN_TCB *task);
```

Разрешен вызов:

В контексте задачи

Параметры функции:

task

указатель на TCB восстанавливаемой задачи

Возвращаемые значения:

TERR_WRONG_PARAM

некорректное значение параметра (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

TERR_NOEXS

попытка восстановления объекта, не являющегося задачей (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

TERR_WCONTEXT

попытка восстановления задачи в прерывании или в пользовательской критической секции

TERR_WSTATE

попытка восстановления задачи, которая находится в состоянии, отличном от *WAITING_SUSPENDED* или *SUSPENDED*

TERR_NO_ERR

успешное выполнение

Пример вызова:

```
extern TN_TCB tcb_Task1  TN_DATA;

/* ... */
if (need_resume_task_1)
    tn_task_resume(&tcb_Task1);
/* ... */
```

tn_task_iresume()

Функция предназначена для вывода задачи из состояния *SUSPENDED* в прерывании.

Если задача находится в состоянии *SUSPENDED*, она переводится в состояние *READY*, при этом она будет поставлена в очередь готовых к выполнению последней. Если задача находится в состоянии *WAITING_SUSPENDED*, она будет переведена в состояние *WAITING* и продолжит ожидание события.

Вызов:

```
TN_RETVAL tn_task_iresume (TN_TCB *task);
```

Разрешен вызов:

В прерывании

Параметры функции:

task

указатель на TCB восстанавливаемой задачи

Возвращаемые значения:

TERR_WRONG_PARAM некорректное значение параметра (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

TERR_NOEXS попытка восстановления объекта, не являющегося задачей (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

TERR_WCONTEXT попытка восстановления задачи в контексте задачи или в пользовательской критической секции

TERR_WSTATE попытка восстановления задачи, которая находится в состоянии, отличном от *WAITING_SUSPENDED* или *SUSPENDED*

TERR_NO_ERR успешное выполнение

Пример вызова:

```
extern TN_TCB tcb_Task1 TN_DATA;

tn_sys_interrupt (_T3Interrupt)    /* прерывание от таймера */
{
    IFS0bits.T2IF = 0;             /* сброс флага прерывания */
    tn_task_iresume(&tcb_Task1);    /* восстановление задачи tcb_Task1 */
}
```

9.1.4. Приостановка выполнения и пробуждение задачи

`tn_task_sleep()`

Функция переводит текущую задачу в ожидание на время не меньше чем `timeout` системных тиков. Если время ожидания истекло, а задача не была восстановлена (сервисом `tn_task_wakeup()`), задача будет выведена планировщиком из состояния ожидания и продолжит выполнение когда станет наиболее приоритетной.

Задача может быть переведена в ожидание с параметром `TN_WAIT_INFINITE`. В этом случае задача будет находится в ожидании до тех пор, пока не будет вызван сервис `tn_task_wakeup()` или `tn_task_iwakeup()`.

Каждая задача имеет счетчик запросов на пробуждение. Если у текущей задачи этот счетчик больше или равен 1, то вызов сервиса `tn_task_sleep()` декрементирует счетчик, а задача продолжает выполнение.

Вызов:

```
TN_RETVAL tn_task_sleep (TN_TIMEOUT timeout);
```

Разрешен вызов:

В контексте задачи

Параметры функции:

`timeout`

интервал времени в системных тиках на который задача будет переведена в ожидание. `timeout` \in `[1..TN_WAIT_INFINITE]`, где `TN_WAIT_INFINITE` = 65'535 для 16-битных контроллеров и `TN_WAIT_INFINITE` = 4'294'967'295 для 32-битных контроллеров

Возвращаемые значения:

`TERR_WRONG_PARAM` некорректное значение параметра, `timeout` = 0 (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

`TERR_WCONTEXT` попытка вызова сервиса в прерывании или в пользовательской критической секции

`TERR_NO_ERR` успешное выполнение

Пример вызова:

```
void TN_TASK Task1 (void *param)
{
    for (;;)
    {
        /* ... */
        tn_task_sleep(10);
        /* ... */
    }
}
```

[tn_task_wakeup\(\)](#)

Функция предназначена для пробуждения приостановленной с помощью функции [tn_task_sleep\(\)](#) задачи. При этом задача продолжит выполнение с места возврата из функции [tn_task_sleep\(\)](#) без ошибок.

Если функция пытается пробудить задачу, которая еще не приостановлена, счетчик запросов на пробуждение будет увеличен на 1.

Вызов:

```
TN_RETVAL tn_task_wakeup (TN_TCB *task);
```

Разрешен вызов:

В контексте задачи

Параметры функции:

`task`

указатель на TCB пробуждаемой задачи

Возвращаемые значения:

`TERR_WRONG_PARAM` некорректное значение параметра (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

`TERR_NOEXS` попытка пробуждения объекта, не являющегося задачей (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

`TERR_WCONTEXT`

попытка пробуждения задачи в прерывании или в пользовательской критической секции

попытка пробуждения задачи, находящейся в состоянии *DORMANT*

`TERR_OVERFLOW` счетчик запросов на пробуждение переполнен (=1)

`TERR_NO_ERR` спешное выполнение

Пример вызова:

```
extern TN_TCB tcb_Task1 TN_DATA;

/* ... */
if (need_wakeup_task_1)
    tn_task_wakeup(&tcb_Task1);
/* ... */
```

tn_task_iwakeup()

Функция предназначена для пробуждения приостановленной с помощью функции [tn_task_sleep\(\)](#) задачи в прерывании. При этом задача продолжит выполнение с места возврата из функции [tn_task_sleep\(\)](#) без ошибок.

Если функция пытается пробудить задачу, которая еще не приостановлена, счетчик запросов на пробуждение будет увеличен на 1.

Вызов:

```
TN_RETVAL tn_task_wakeup (TN_TCB *task);
```

Разрешен вызов:

В прерывании

Параметры функции:

task

указатель на TCB пробуждаемой задачи

Возвращаемые значения:

TERR_WRONG_PARAM некорректное значение параметра (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

TERR_NOEXS попытка пробуждения объекта, не являющегося задачей (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

TERR_WCONTEXT

попытка пробуждения задачи в контексте задачи или в пользовательской критической секции

попытка пробуждения задачи, находящейся в состоянии *DORMANT*

TERR_OVERFLOW счетчик запросов на пробуждение переполнен (=1)

TERR_NO_ERR спешное выполнение

Пример вызова:

```
extern TN_TCB tcb_Task1  TN_DATA;

tn_sys_interrupt (_T3Interrupt)    /* прерывание от таймера */
{
    IFS0bits.T2IF = 0;            /* сброс флага прерывания */
    tn_task_iwakeup(&tcb_Task1);   /* активация задачи tcb_Task1 */
}
```

9.1.5. Форсированный вывод задачи из состояния WAITING

tn_task_release_wait()

Функция выводит задачу из состояния ожидания события вне зависимости от причины ожидания.

Если задача находится в состоянии *WAITING*, она переводится в состояние *READY*. Если задача находится в состоянии *WAITING_SUSPEND*, она переводится в состояние *SUSPEND*. Если задача находилась в состоянии *WAITING* по причине вызова функции *tn_task_sleep()*, вызов функции *tn_task_release_wait()* с указателем на эту задачу кроме всего прочего сбрасывает счетчик попыток пробуждения задачи. Функция не может использоваться для активации задачи, которая находится в состоянии *SUSPEND*. Задача не может вызывать функцию *tn_task_release_wait()* с указателем на саму себя.

Вызов:

```
TN_RETURN tn_task_release_wait (TN_TCB *task);
```

Разрешен вызов:

В контексте задачи

Параметры функции:

task

указатель на TCB задачи

Возвращаемые значения:

TERR_WRONG_PARAM некорректное значение параметра (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

TERR_NOEXS попытка пробуждения объекта, не являющегося задачей

TERR_WCONTEXT

попытка пробуждения задачи в прерывании или в пользовательской критической секции

задача находится в состоянии, отличном от *WAITING* или *WAITING_SUSPEND*

TERR_NO_ERR успешное выполнение

Пример вызова:

```
extern TN_TCB tcb_Task1 TN_DATA;

/* ... */
if (need_force_task_1)
    tn_task_release_wait(&tcb_Task1);
/* ... */
```

tn_task_irelease_wait()

Функция выводит задачу из состояния ожидания события вне зависимости от причины ожидания. Функция может быть вызвана только в прерывании.

Если задача находится в состоянии *WAITING*, она переводится в состояние *READY*. Если задача находится в состоянии *WAITING_SUSPEND*, она переводится в состояние *SUSPEND*. Если задача находилась в состоянии *WAITING* по причине вызова функции `tn_task_sleep()`, вызов функции `tn_task_irelease_wait()` с указателем на эту задачу кроме всего прочего сбрасывает счетчик попыток пробуждения задачи. Функция не может использоваться для активации задачи, которая находится в состоянии *SUSPEND*.

Вызов:

```
TN_RETVAL tn_task_irelease_wait (TN_TCB *task);
```

Разрешен вызов:

В прерывании

Параметры функции:

`task`

указатель на TCB задачи

Возвращаемые значения:

`TERR_WRONG_PARAM`

некорректное значение параметра (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

`TERR_NOEXS`

попытка пробуждения объекта, не являющегося задачей

`TERR_WCONTEXT`

попытка пробуждения задачи в пользовательской критической секции или контексте задачи

задача находится в состоянии, отличном от *WAITING* или *WAITING_SUSPEND*

`TERR_NO_ERR`

успешное выполнение

Пример вызова:

```
extern TN_TCB tcb_Task1  TN_DATA;

tn_sys_interrupt (_T3Interrupt)    /* прерывание от таймера */
{
    IFS0bits.T2IF = 0;             /* сброс флага прерывания */
    tn_task_irelease_wait(&tcb_Task1); /* восстановление задачи tcb_Task1 */
}
```

9.1.6. Изменение приоритета задачи

tn_task_change_priority()

Функция изменяет натуральный (заданный при создании) приоритет задачи.

Функция может изменять приоритет как текущей, так и любой другой задачи, которая находится в состоянии отличном от *DORMANT*.

Если после изменения приоритета текущей задачи она становится менее приоритетной чем одна из задач, готовая к выполнению, то запускается более приоритетная задача.

Если после изменения приоритета задачи готовой к выполнению, она становится наиболее приоритетной, то она запускается - становится активной.

Изменение приоритета задачи, находящейся в состоянии останова или ожидания не меняет состояния задачи.

Вызов:

`TN_RETVAL tn_task_change_priority (TN_TCB *task, TN_UWORD new_priority);`

Разрешен вызов:

В контексте задачи

Параметры функции:

`task`

указатель на TCB задачи, изменяющей приоритет. Задача может изменить свой приоритет.

`new_priority`

новый приоритет задачи от 1 до 14 для 16-битных контроллеров и от 1 до 30 - для 32-битных контроллеров. Если значение параметра равно 0, то задача восстанавливает базовый приоритет, назначенный при создании.

Возвращаемые значения:

`TERR_WRONG_PARAM` некорректное значение параметра (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

`TERR_NOEXS` попытка изменения приоритета объекта, не являющегося задачей

`TERR_WCONTEXT`

попытка изменения приоритета задачи в прерывании или в пользовательской критической секции

попытка изменения приоритета задачи, находящейся в состоянии *DORMANT*

`TERR_NO_ERR` успешное выполнение

Пример вызова:

`tn_task_change_priority(&myTask, 10);`

9.1.7. Получение информации о задаче

[tn_task_reference\(\)](#)

Функция предназначена для получения информации о задаче, такой как текущее состояние, причина ожидания, время таймаута, текущий приоритет и др. После вызова функции заполняется структура типа `TN_TASK_REF`, указатель на которую передается в качестве параметра функции:

```
typedef struct _TN_TASK_REF
{
    TN_TASK_STATE      state;
    TN_TASK_WAIT_REASON wait_reason;
    TN_UWORD           base_priority;
    TN_UWORD           current_priority;
    TN_TIMEOUT          timeout;
} TN_TASK_REF;
```

В состав структура `TN_TASK_REF` входят следующие элементы:

`state`

состояние задачи, может принимать одно из следующих значений:

`TSK_STATE_RUNNABLE` Задача находится в очереди готовых к выполнению

`TSK_STATE_WAIT` Задача ожидает событие

`TSK_STATE_SUSPEND` Задача приостановлена

`TSK_STATE_DORMANT` Задача создана, но еще не запущена

`wait_reason`

событие, которого ожидает задача, может принимать одно из следующих значений:


`TSK_WAIT_REASON_SLEEP` Задача ожидает таймаута, переведена в состояние ожидания функцией `tn_task_sleep()`


`TSK_WAIT_REASON_SEM` Задача ожидает освобождения семафора

`TSK_WAIT_REASON_EVENT` Задача ожидает флаг

`TSK_WAIT_REASON_DQUE_WSEND` Задача ожидает освобождение очереди сообщений

`TSK_WAIT_REASON_DQUE_WRECEIVE` Задача ожидает сообщения

`TSK_WAIT_REASON_MUTEX_C` Задача ожидает освобождения ресурса, заблокированного "priority ceiling" мьютексом 

`TSK_WAIT_REASON_MUTEX_C_BLK` Задача ожидает освобождения ресурса, заблокированного "priority ceiling" мьютексом 

`TSK_WAIT_REASON_MUTEX_I` Задача ожидает освобождения ресурса, заблокированного "priority inheritance" мьютексом

`TSK_WAIT_REASON_WFIXMEM` Задача ожидает освобождения блока памяти фиксированного размера

base_priority

базовый приоритет задачи (назначенный при ее создании)

current_priority

текущий приоритет задачи

timeout

время в системных тиках до момента перевода задачи в состояние `TSK_STATE_RUNNABLE` (актуально в случае, если задача находится в состоянии ожидания с таймаутом)

Вызов:

```
TN_RETVAL tn_task_reference(TN_TCB *task, TN_TASK_REF *ref);
```

Разрешен вызов:

В контексте задачи

Параметры функции:

task

указатель на TCB задачи

ref

указатель на структуру информации о задаче

Возвращаемые значения:

`TERR_WRONG_PARAM` некорректное значение параметра (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

`TERR_NOEXS` задача не существует (еще не создана)

`TERR_WCONTEXT` попытка вызова сервиса в прерывании или в пользовательской критической секции

`TERR_NO_ERR` успешное выполнение

Пример вызова:

```
TN_TASK_REF task_info;

if (tn_task_reference(&task_1, &task_info) == TERR_NO_ERR)
{
    if (task_info.state == TSK_STATE_WAIT &&
        task_info.wait_reason == TSK_WAIT_REASON_SLEEP
    )
    {
        tn_task_wakeup(&task_1);
    }
}
if (tn_task_reference(&task_1, &task_info) == TERR_NOEXS)
{
    /* задача не создана */
}
```

Функция предназначена для получения информации о задаче в прерывании. После вызова функции заполняется структура типа `TN_TASK_REF`, указатель на которую передается в качестве параметра функции:

```
typedef struct _TN_TASK_REF
{
    TN_TASK_STATE      state;
    TN_TASK_WAIT_REASON wait_reason;
    TN_UWORD           base_priority;
    TN_UWORD           current_priority;
    TN_TIMEOUT         timeout;
} TN_TASK_REF;
```

В состав структура `TN_TASK_REF` входят следующие элементы:

`state`

состояние задачи, может принимать одно из следующих значений:

`TSK_STATE_RUNNABLE` Задача находится в очереди готовых к выполнению

`TSK_STATE_WAIT` Задача ожидает событие

`TSK_STATE_SUSPEND` Задача приостановлена

`TSK_STATE_DORMANT` Задача создана, но еще не запущена

`wait_reason`

событие, которого ожидает задача, может принимать одно из следующих значений:


`TSK_WAIT_REASON_SLEEP` Задача ожидает таймаута, переведена в состояние ожидания функцией `tn_task_sleep()`


`TSK_WAIT_REASON_SEM` Задача ожидает освобождения семафора

`TSK_WAIT_REASON_EVENT` Задача ожидает флаг

`TSK_WAIT_REASON_DQUE_WSEND` Задача ожидает освобождение очереди сообщений

`TSK_WAIT_REASON_DQUE_WRECEIVE` Задача ожидает сообщения

`TSK_WAIT_REASON_MUTEX_C` Задача ожидает освобождения ресурса, заблокированного "priority ceiling" мютексом 

`TSK_WAIT_REASON_MUTEX_C_BLK` Задача ожидает освобождения ресурса, заблокированного "priority ceiling" мютексом 

`TSK_WAIT_REASON_MUTEX_I` Задача ожидает освобождения ресурса, заблокированного "priority inheritance" мютексом

`TSK_WAIT_REASON_WFIXMEM` Задача ожидает освобождения блока памяти фиксированного размера

`base_priority`

базовый приоритет задачи (назначенный при ее создании)

current_priority

текущий приоритет задачи

timeout

время в системных тиках до момента перевода задачи в состояние `TSK_STATE_RUNNABLE` (актуально в случае, если задача находится в состоянии ожидания с таймаутом)

Вызов:

```
TN_RETVAL tn_task_ireference(TN_TCB *task, TN_TASK_REF *ref);
```

Разрешен вызов:

В прерывании

Параметры функции:

task

указатель на TCB задачи

ref

указатель на структуру информации о задаче

Возвращаемые значения:

`TERR_WRONG_PARAM`

некорректное значение параметра (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

`TERR_NOEXS`

задача не существует (еще не создана)

`TERR_WCONTEXT`

попытка вызова сервиса в контексте задачи

`TERR_NO_ERR`

успешное выполнение

Пример вызова:

```
extern TN_TCB tcb_Task1 TN_DATA;
TN_TASK_REF task_info;

tn_sys_interrupt (_T2Interrupt)    /* прерывание от таймера */
{
    IFS0bits.T2IF = 0;             /* сброс флага прерывания */
    if (tn_task_ireference(&tcb_Task1, &task_info) == TERR_NO_ERR)
    {
        if (task_info.state == TSK_STATE_SUSPEND)
            tn_task_iresume(&tcb_Task1);
    }
}
```

9.2. Сервисы управления семафорами

9.2.1. Создание и удаление семафора

tn_sem_create()

Функция предназначена для создания семафора. Поле `id_sem` структуры `sem` должно быть равно нулю до момента создания семафора. Таким образом уже созданные семафоры защищаются от повторного создания.

Память для управляющей структуры `sem` должна быть выделена до момента создания семафора. Память может быть выделена на этапе компиляции (объявление глобальной переменной типа `TN_SEM`), либо динамически, если пользовательское приложение использует менеджер памяти.

Вызов:

```
TN_RETVAL tn_sem_create(TN_SEM *sem, TN_UWORD start_value, TN_UWORD max_val);
```

Разрешен вызов:

В контексте задачи

Параметры функции:

`sem`

указатель на структуру семафора типа `TN_SEM`. Структура должна быть создана до момента вызова функции, статически или динамически

`start_value`

начальное значения счетчика свободных ресурсов семафора. Если этот параметр равен 0, то семафор считается занятым на момент создания.

`max_val`

максимальное значение счетчика свободных ресурсов семафора. Если этот параметр равен 1, то создается двоичный семафор.

Возвращаемые значения:

`TERR_WRONG_PARAM` некорректное значение параметра (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

`TERR_EXS` попытка создания семафора, который уже создан

`TERR_WCONTEXT` попытка создания семафора в прерывании или в пользовательской критической секции

`TERR_NO_ERR` успешное выполнение

Пример вызова:

```
TN_SEM sem_test;  
  
tn_sem_create(&sem_test, 1, 1); /* создается свободный бинарный семафор */
```

tn_sem_delete()

Функция предназначена для удаления семафора. Поле `id_sem` структуры `sem` после выполнения сервиса устанавливается в 0. Все задачи, ожидающие семафор, выйдут из сервиса ожидания с кодом возврата `TERR_DLT`.

Вызов:

```
TN_RETVAL tn_sem_delete (TN_SEM *sem);
```

Разрешен вызов:

В контексте задачи

Параметры функции:

`sem`

указатель на структуру семафора типа `TN_SEM`.

Возвращаемые значения:

`TERR_WRONG_PARAM`

некорректное значение параметра (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

`TERR_NOEXS`

попытка удалить объект, который не является семафором

`TERR_WCONTEXT`

попытка удаления семафора в прерывании или в пользовательской критической секции

`TERR_NO_ERR`

успешное выполнение

Пример вызова:

```
extern TN_SEM sem_test;  
  
tn_sem_delete(&sem_test);
```

9.2.2. Освобождение семафора

tn_sem_signal()

Сервис увеличивает счетчик свободных ресурсов семафора или, другими словами, *освобождает семафор*.

Если семафор был занят, то задача, стоящая первой в очереди ожидающих семафор, активируется. Счетчик ресурсов семафора при этом не меняется.

Если очередь задач ожидающих семафор пуста, и счетчик свободных ресурсов меньше максимального значения, то он увеличивается на единицу. Если счетчик свободных ресурсов равен максимальному значению, то он не увеличивается и сервис возвращает код `TERR_OVERFLOW`.

Вызов:

`TN_RETVAL tn_sem_signal (TN_SEM *sem);`

Разрешен вызов:

В контексте задачи

Параметры функции:

`sem`

указатель на структуру семафора

Возвращаемые значения:

`TERR_WRONG_PARAM`

некорректное значение параметра (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

`TERR_NOEXS`

попытка освобождения объекта, не являющегося семафором.

`TERR_WCONTEXT`

попытка освобождения семафора в прерывании или в пользовательской критической секции

`TERR_OVERFLOW`

счетчик свободных ресурсов достиг максимального значения

`TERR_NO_ERR`

успешное выполнение

Пример вызова:

`TN_SEM sem_test;`

`tn_sem_signal(&sem_test);`

tn_sem_isignal()

Сервис увеличивает счетчик свободных ресурсов семафора или, другими словами, *освобождает семафор* в прерывании.

Если семафор был занят, то задача, стоящая первой в очереди ожидающих семафор, активируется после выхода из прерывания. Счетчик ресурсов семафора при этом не меняется.

Если очередь задач ожидающих семафор пуста, и счетчик свободных ресурсов меньше максимального значения, то он увеличивается на единицу. Если счетчик свободных ресурсов равен максимальному значению, то он не увеличивается и сервис возвращает код `TERR_OVERFLOW`.

Вызов:

```
TN_RETVAL tn_sem_isignal (TN_SEM *sem);
```

Разрешен вызов:

В прерывании

Параметры функции:

`sem`

указатель на структуру семафора

Возвращаемые значения:

`TERR_WRONG_PARAM` некорректное значение параметра (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

`TERR_NOEXS` попытка освобождения объекта, не являющегося семафором.

`TERR_WCONTEXT` попытка освобождения семафора в контексте задачи

`TERR_OVERFLOW` счетчик свободных ресурсов достиг максимального значения

`TERR_NO_ERR` успешное выполнение

Пример вызова:

```
extern TN_SEM sem_test;

tn_sys_interrupt (_T2Interrupt)    /* прерывание от таймера */
{
    IFS0bits.T2IF = 0;             /* сброс флага прерывания */
    tn_sem_isignal(&sem_test);     /* освобождение семафора sem_test */
}
```


9.2.3. Захват семафора

tn_sem_acquire()

Сервис предназначен для захвата семафора.

Если счетчик свободных ресурсов семафора больше нуля (семафор свободен), то задача, которая пытается захватить семафор продолжает выполнение, а счетчик ресурсов уменьшается на единицу.

Если счетчик свободных ресурсов семафора равен нулю (семафор захвачен), то задача, которая пытается захватить семафор ставится в конец очереди задач ожидающих семафор и переводится в состояние ожидания. Счетчик ресурсов семафора не меняется.

Параметр `timeout` задает время ожидания семафора в системных тиках. Если в течении этого времени семафор не будет захвачен задачей, она удаляется из очереди ожидающих семафор и переводится в состояние готовых к выполнению. Сервис возвращает значение `TERR_TIMEOUT`.

Если значение параметра `timeout` равно `TN_WAIT_INFINITE`, то будет ожидать семафор до тех пор, пока он не освободиться.

Вызов:

`TN_RETVAL tn_sem_acquire (TN_SEM *sem, TN_TIMEOUT timeout);`

Разрешен вызов:

В контексте задачи

Параметры функции:

<code>sem</code>	казатель на структуру семафора
<code>timeout</code>	таймаут в течение которого задача ожидает семафор

Возвращаемые значения:

`TERR_WRONG_PARAM` некорректное значение параметра (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

`TERR_NOEXS` попытка захвата объекта, не являющегося семафором.

`TERR_WCONTEXT` попытка захвата семафора в прерывании или в пользовательской критической секции

`TERR_TIMEOUT` выход из сервиса по таймауту

`TERR_DLT` выход из сервиса, так как ожидаемый семафор удален

`TERR_NO_ERR` успешное выполнение

Пример вызова:

`TN_SEM sem_test;

tn_sem_acquire(&sem_test, 10);`

Сервис предназначен для захвата одного ресурса семафора без перевода задачи в состояние ожидания.

Если счетчик свободных ресурсов семафора больше нуля (семафор свободен), он уменьшается на единицу. Если счетчик свободных ресурсов семафора равен нулю (семафор захвачен), сервис возвращает код `TERR_TIMEOUT`. Счетчик ресурсов семафора не меняется. В любом случае задача не блокируется.

Вызов:

```
TN_RETVAL tn_sem_polling (TN_SEM *sem);
```

Разрешен вызов:

В контексте задачи

Параметры функции:

`sem`

указатель на структуру семафора

Возвращаемые значения:

`TERR_WRONG_PARAM`

некорректное значение параметра (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

`TERR_NOEXS`

попытка захвата объекта, не являющегося семафором.

`TERR_WCONTEXT`

попытка захвата семафора в прерывании или в пользовательской критической секции

`TERR_TIMEOUT`

семафор уже захвачен

`TERR_NO_ERR`

успешное выполнение

Пример вызова:

```
TN_SEM  sem_test;

if (tn_sem_polling(&sem_test) == TERR_NO_ERR)
{
    /* ... */
}
```

Сервис предназначен для захвата одного ресурса семафора в прерывании.

Если счетчик свободных ресурсов семафора больше нуля (семафор свободен), он уменьшается на единицу. Если счетчик свободных ресурсов семафора равен нулю (семафор захвачен), сервис возвращает код `TERR_TIMEOUT`, а счетчик ресурсов семафора не меняется.

Вызов:

```
TN_RETVAL tn_sem_ipolling (TN_SEM *sem);
```

Разрешен вызов:

В прерывании

Параметры функции:

`sem`

указатель на структуру семафора

Возвращаемые значения:

`TERR_WRONG_PARAM` некорректное значение параметра (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

`TERR_NOEXS` попытка захвата объекта, не являющегося семафором.

`TERR_WCONTEXT` попытка захвата семафора в контексте задачи

`TERR_TIMEOUT` семафор уже захвачен

`TERR_NO_ERR` успешное выполнение

Пример вызова:

```
extern TN_SEM sem_test;

tn_sys_interrupt (_T2Interrupt)    /* прерывание от таймера */
{
    IFS0bits.T2IF = 0;             /* сброс флага прерывания */
    if (tn_task_ipolling(&sem_test) == TERR_NO_ERR)
    {
        /* ... */
    }
}
```

9.3. Сервисы управления флагами

9.3.1. Создание и удаление флага

[tn_event_create\(\)](#)

Функция предназначена для создания флага. Поле `id_event` структуры `evf` должно быть равно нулю до момента создания флага. Таким образом уже созданные флаги защищаются от повторного создания.

Память для управляющей структуры `evf` должна быть выделена до момента создания флага. Память может быть выделена на этапе компиляции (объявление глобальной переменной типа `TN_EVENT`), либо динамически, если пользовательское приложение использует менеджер памяти.

Параметр `attr` определяет тип флага. Если параметр равен `TN_EVENT_ATTR_MULTI`, то флаг может ожидать несколько задач. Если параметр равен `TN_EVENT_ATTR_SINGLE` - флаг может ожидать только одна задача. В этом случае допустимо объявление параметра `TN_EVENT_ATTR_CLR`, который указывает на то, что битовая маска будет сбрасываться автоматически. Параметр `TN_EVENT_ATTR_CLR` объявляется вместе с параметром `TN_EVENT_ATTR_SINGLE`:
(`TN_EVENT_ATTR_SINGLE | TN_EVENT_ATTR_CLR`).

Вызов:

```
TN_RETVAL tn_event_create (TN_EVENT *evf, TN_UWORD attr, TN_UWORD pattern);
```

Разрешен вызов: В контексте задачи, в пользовательской критической секции, в прерывании

Параметры функции:

`evf` указатель на структуру флага типа `TN_EVENT`. Структура должна быть создана до момента вызова функции, статически или динамически

`attr` тип флага:

<code>TN_EVENT_ATTR_MULTI</code>	флаг может ожидать несколько задач
<code>TN_EVENT_ATTR_SINGLE</code>	флаг может ожидать только одна задача
<code>TN_EVENT_ATTR_CLR</code>	битовая маска будет сброшена автоматически. Определение может быть объединено по ИЛИ только с <code>TN_EVENT_ATTR_SINGLE</code>

`pattern` значение битовой маски сразу после создания флага

Возвращаемые значения:

`TERR_WRONG_PARAM` некорректное значение параметра (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

`TERR_EXS` попытка создания флага, который уже создан

`TERR_NO_ERR` успешное выполнение

Пример вызова:

```
TN_EVENT event_test;

/* Создается флаг, который может ожидать только одна задача с
   автоматическим сбросом соответствующих битов в битовой
   маске. Начальное значение битовой маски равно 0x5555
*/
tn_event_create(&event_test, TN_EVENT_ATTR_SINGLE | TN_EVENT_ATTR_CLR, 0x5555);
```

`tn_event_delete()`

Функция предназначена для удаления флага. Поле `id_event` структуры `evf` после выполнения сервиса устанавливается в 0. Все задачи, ожидающие флаг, выйдут из сервиса ожидания с кодом возврата `TERR_DLT`.

Вызов:

```
TN_RETVAL tn_event_delete (TN_EVENT *evf);
```

Разрешен вызов:

В контексте задачи

Параметры функции:

`evf`

указатель на структуру удаляемого флага

Возвращаемые значения:

`TERR_WRONG_PARAM`

некорректное значение параметра (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

`TERR_WCONTEXT`

попытка удаления флага в пользовательской критической секции или в прерывании

`TERR_EXS`

попытка удаления объекта, не являющегося флагом

`TERR_NO_ERR`

успешное выполнение

Пример вызова:

```
TN_EVENT event_test;  
  
tn_event_delete(&event_test);
```

9.3.2. Установка и сброс битовой маски флага

`tn_event_set()`

Функция предназначена для установки битов в битовой маске флага `evf`. Биты устанавливаются по логическому ИЛИ с параметром-маской `pattern`.

После того как битовая маска флага обновлена, проверяются все задачи, ожидающие флаг. Если условие для запуска одной из задачи соответствует битовой маске, задача переводится в состояние готовых к выполнению.

Если флаг имеет атрибут `TN_EVENT_ATTR_MULTI`, то в состояние готовых к выполнению переводятся все задачи, условие для запуска которых соответствует битовой маске.

Если флаг имеет атрибут `TN_EVENT_ATTR_CLR`, то битовая маска флага сбрасывается.

Вызов:

```
TN_RETURN tn_event_set (TN_EVENT *evf, TN_UWORD pattern);
```

Разрешен вызов:

В контексте задачи

Параметры функции:

`evf`

указатель на объект типа флаг

`pattern`

устанавливаемые биты в битовой маске флага. Например, если битовая маска флага до вызова сервиса была равно `0b0011001101010101`, а параметр `pattern` равен `0b1100000000000000`, то битовая маска флага станет равна `0b1111001101010101`

Возвращаемые значения:

`TERR_WRONG_PARAM` некорректное значение параметра (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

`TERR_NOEXS` попытка изменения объекта, не являющегося флагом (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

`TERR_WCONTEXT` попытка изменения флага в пользовательской критической секции или в прерывании

`TERR_NO_ERR` успешное выполнение

Пример вызова:

```
TN_EVENT event_test;  
  
tn_event_set(&event_test, 0x8000);
```

tn_event_iset()

Функция предназначена для установки битов в битовой маске флага `evf` в прерывании. Биты устанавливаются по логическому ИЛИ с параметром-маской `pattern`.

После того как битовая маска флага обновлена, проверяются все задачи, ожидающие флаг. Если условие для запуска одной из задачи соответствует битовой маске, задача переводится в состояние готовых к выполнению.

Если флаг имеет атрибут `TN_EVENT_ATTR_MULTI`, то в состояние готовых к выполнению переводятся все задачи, условие для запуска которых соответствует битовой маске.

Если флаг имеет атрибут `TN_EVENT_ATTR_CLR`, то битовая маска флага сбрасывается.

Вызов:

```
TN_RETVAL tn_event_iset (TN_EVENT *evf, TN_UWORD pattern);
```

Разрешен вызов: В прерывании

Параметры функции:

`evf` указатель на объект типа флаг

`pattern` устанавливаемые биты в битовой маске флага. Например, если битовая маска флага до вызова сервиса была равно `0b0011001101010101`, а параметр `pattern` равен `0b1100000000000000`, то битовая маска флага станет равна `0b1111001101010101`

Возвращаемые значения:

`TERR_WRONG_PARAM` некорректное значение параметра (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

`TERR_NOEXS`

попытка изменения объекта, не являющегося флагом (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

`TERR_WCONTEXT`

попытка вызова сервиса в контексте задачи

`TERR_NO_ERR`

успешное выполнение

Пример вызова:

```
extern TN_EVENT event;

tn_sys_interrupt (_T2Interrupt)     /* прерывание от таймера */
{
    IFS0bits.T2IF = 0;               /* сброс флага прерывания */
    tn_event_iset(&event, 0x8000);
}
```

`tn_event_clear()`

Функция предназначена для сброса битов в битовой маске флага `evf`. Биты сбрасываются по логическому И с параметром-маской `pattern`.

Этот сервис не предназначен для запуска задач, ожидающих определенную битовую маску. Его можно использовать для сброса события в задаче, ожидавшей флаг.

Вызов:

```
TN_RETVAL tn_event_clear (TN_EVENT *evf, TN_UWORD pattern);
```

Разрешен вызов:

В контексте задачи

Параметры функции:

`evf`

указатель на объект типа флаг

`pattern`

сбрасываемые биты в битовой маске флага. *Замечание:* сбрасываемые биты должны быть равны 0, параметр `pattern` не должен принимать значения `0xFFFF`. Например, если битовая маска флага до вызова сервиса была равна `0b0011001101010101`, а параметр `pattern` равен `0b1100111111111111`, то битовая маска флага станет равна `0b0000001101010101`

Возвращаемые значения:

`TERR_WRONG_PARAM`

некорректное значение параметра (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

`TERR_NOEXS`

попытка изменения объекта, не являющегося флагом (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

`TERR_WCONTEXT`

попытка вызова сервиса в пользовательской критической секции или в прерывании

`TERR_NO_ERR`

успешное выполнение

Пример вызова:

```
TN_EVENT event;  
tn_event_clear(&event, ~(0x8000));
```


tn_event_iclear()

Функция предназначена для сброса битов в битовой маске флага `evf` в прерывании. Биты сбрасываются по логическому И с параметром-маской `pattern`.

Этот сервис не предназначен для запуска задач, ожидающих определенную битовую маску. Его можно использовать для сброса (отмены) события в прерывания.

Вызов:

```
TN_RETVAL tn_event_iclear (TN_EVENT *evf, TN_UWORD pattern);
```

Разрешен вызов:

В прерывании

Параметры функции:

`evf`

указатель на объект типа флаг

`pattern`

сбрасываемые биты в битовой маске флага. **Замечание:** сбрасываемые биты должны быть равны 0, параметр `pattern` не должен принимать значения `0xFFFF`. Например, если битовая маска флага до вызова сервиса была равна `0b0011001101010101`, а параметр `pattern` равен `0b1100111111111111`, то битовая маска флага станет равна `0b0000001101010101`

Возвращаемые значения:

`TERR_WRONG_PARAM`

некорректное значение параметра ¹⁾

`TERR_NOEXS`

попытка изменения объекта, не являющегося флагом ²⁾

`TERR_WCONTEXT`

попытка вызова сервиса в контексте задачи

`TERR_NO_ERR`

успешное выполнение

Пример вызова:

```
extern TN_EVENT event;

tn_sys_interrupt (_T2Interrupt)    /* прерывание от таймера */
{
    IFS0bits.T2IF = 0;             /* сброс флага прерывания */
    tn_event_iclear(&event, ~(0x8000));
}
```

9.3.3. Ожидание флага

tn_event_wait()

Функция предназначена для перевода вызвавшей ее задачи в состояние ожидания до тех пор, пока условие совпадения с битовой маской флага `evf` не будет выполнено. Условие определяется параметром `wait_pattern` и режимом ожидания `wait_mode`. Как только условие будет выполнено, битовая маска флага будет возвращена по указателю `p_flags_pattern`.

Если на момент вызова функции условие выполняется, задача не будет переведена в состояние ожидания и сервис завершит свое выполнение.

Если на момент вызова функции условие не выполняется, то задача переводится в состояние ожидания и ставится в очередь ожидания флага. Функция может быть вызвана с таймаутом - если значение параметра `timeout` не равно `TN_WAIT_INFINITE`, то по прошествии `timeout` системных тиков, функция вернет управление прерванной задаче с кодом возврата `TERR_TIMEOUT`.

Если флаг `evf` имеет атрибут `TN_EVENT_ATTR_SINGLE`, а очередь ожидания флага не пуста, то функция вернет код ошибки `TERR_ILUSE`, что означает попытку ожидания флага, предназначенного только для одной (уже ожидающей его) задачи.

Если флаг `evf` имеет атрибут `TN_EVENT_ATTR_CLR`, битовая маска флага обнуляется.

Параметр `wait_mode` формирует условие ожидания флага. Если `wait_mode == TN_EVENT_WCOND_OR`, то условие ожидания будет выполнено, если хотя бы один бит из битовой маски флага будет соответствовать `wait_pattern`. Если `wait_mode == TN_EVENT_WCOND_AND`, то для выполнения условия ожидания необходимо чтобы все биты битовой маски флага соответствовали параметру `wait_pattern`.

Вызов:

```
TN_RETVAL tn_event_wait (TN_EVENT *evf,
                        TN_UWORD wait_pattern,
                        TN_UWORD wait_mode,
                        TN_UWORD *p_flags_pattern,
                        TN_TIMEOUT timeout
                        );
```

Разрешен вызов:

В контексте задачи

Параметры функции:

`evf`

указатель на объект типа флаг

`wait_pattern`

параметр сравнения с битовой маской флага

`wait_mode`

режим ожидания, формирующий условие, может принимать одно из двух значений:

`TN_EVENT_WCOND_OR` условие выполняется, если хотя бы один из битов маски флага совпадает с маской сравнения `wait_pattern`

`TN_EVENT_WCOND_AND` условие выполняется только в том случае, если хотя бы все биты маски флага совпадают с маской сравнения `wait_pattern`

`p_flags_pattern`

указатель по которому возвращается значение битовой маски флага после выполнения условия

`timeout`

таймаут, в течении которого задача ожидает выполнения условия

Возвращаемые значения:

TERR_WRONG_PARAM

некорректное значение параметра [1\)](#)

TERR_NOEXS

попытка обращения к объекту, не являющегося флагом [2\)](#)

TERR_WCONTEXT

попытка изменения флага в пользовательской критической секции или в прерывании

TERR_ILUSE

попытка ожидания флага с атрибутом TN_EVENT_ATTR_SINGLE, если его очередь ожидания не пуста

TERR_TIMEOUT

выход из сервиса по таймауту

TERR_NO_ERR

успешное выполнение

Пример вызова:

```
TN_EVENT event_test;  
TN_UWORD flag;  
  
tn_event_wait(&event_test, 0x8000, TN_EVENT_WCOND_AND, &flag, TN_WAIT_INFINITE);
```

Функция предназначена для проверки битовой маски флага в прерывании.

Условие проверки определяется маской `wait_pattern` и режимом ожидания `wait_mode`. Если условие выполняется, битовая маска флага будет возвращена по указателю `p_flags_pattern`.

Если на момент вызова функции условие не выполняется, функция вернет код ошибки `TERR_TIMEOUT`.

Если флаг `evf` имеет атрибут `TN_EVENT_ATTR_SINGLE`, а очередь ожидания флага не пуста, то функция вернет код ошибки `TERR_ILUSE`, что означает попытку проверки флага, предназначенного только для одной (уже ожидающей его) задачи.

Если флаг `evf` имеет атрибут `TN_EVENT_ATTR_CLR`, битовая маска флага обнуляется.

Параметр `wait_mode` формирует условие совпадения. Если `wait_mode == TN_EVENT_WCOND_OR`, то условие будет выполнено, если хотя бы один бит из битовой маски флага будет соответствовать `wait_pattern`. Если `wait_mode == TN_EVENT_WCOND_AND`, то для выполнения условия необходимо чтобы все биты битовой маски флага соответствовали параметру `wait_pattern`.

Вызов:

```
TN_RETVAL tn_event_iwait (TN_EVENT *evf,
                          TN_UWORD wait_pattern,
                          TN_UWORD wait_mode,
                          TN_UWORD *p_flags_pattern
                          );
```

Разрешен вызов:

В прерывании

Параметры функции:

`evf`

указатель на объект типа флаг

`wait_pattern`

параметр сравнения с битовой маской флага

`wait_mode`

условие совпадения, может принимать одно из двух значений:

`TN_EVENT_WCOND_OR` условие выполняется, если хотя бы один из битов маски флага совпадает с маской сравнения `wait_pattern`

`TN_EVENT_WCOND_AND` условие выполняется только в том случае, если хотя все биты маски флага совпадают с маской сравнения `wait_pattern`

`p_flags_pattern`

указатель по которому возвращается значение битовой маски флага если условие выполнено

Возвращаемые значения:

`TERR_WRONG_PARAM`

некорректное значение параметра ¹⁾

`TERR_NOEXS`

попытка обращения к объекту, не являющегося флагом ²⁾

`TERR_WCONTEXT`

попытка проверки флага в контексте задачи

TERR_ILUSE

попытка проверки флага с атрибутом TN_EVENT_ATTR_SINGLE, если его очередь ожидания не пуста

TERR_TIMEOUT

условие не выполняется

TERR_NO_ERR

успешное выполнение

Пример вызова:

```
extern TN_EVENT event_test;

tn_sys_interrupt (_T2Interrupt)    /* прерывание от таймера */
{
    TN_UWORD flag;

    IFS0bits.T2IF = 0;             /* сброс флага прерывания */
    tn_event_wait(&event_test, 0x8000, TN_EVENT_WCOND_AND, &flag);
}
```

tn_event_wait_polling()

Функция предназначена для проверки битовой маски флага без блокировки задачи.

Условие проверки определяется маской `wait_pattern` и режимом ожидания `wait_mode`. Если условие выполняется, битовая маска флага будет возвращена по указателю `p_flags_pattern`.

Если на момент вызова функции условие не выполняется, функция вернет код ошибки `TERR_TIMEOUT`.

Если флаг `evf` имеет атрибут `TN_EVENT_ATTR_SINGLE`, а очередь ожидания флага не пуста, то функция вернет код ошибки `TERR_ILUSE`, что означает попытку проверки флага, предназначенного только для одной (уже ожидающей его) задачи.

Если флаг `evf` имеет атрибут `TN_EVENT_ATTR_CLR`, битовая маска флага обнуляется.

Параметр `wait_mode` формирует условие совпадения. Если `wait_mode == TN_EVENT_WCOND_OR`, то условие будет выполнено, если хотя бы один бит из битовой маски флага будет соответствовать `wait_pattern`. Если `wait_mode == TN_EVENT_WCOND_AND`, то для выполнения условия необходимо чтобы все биты битовой маски флага соответствовали параметру `wait_pattern`.

Вызов:

```
TN_RETVAL tn_event_wait_polling (TN_EVENT *evf,
                                TN_UWORD wait_pattern,
                                TN_UWORD wait_mode,
                                TN_UWORD *p_flags_pattern
                                );
```

Разрешен вызов:

В контексте задачи

Параметры функции:

`evf`

указатель на объект типа флаг

`wait_pattern`

параметр сравнения с битовой маской флага

`wait_mode`

условие совпадения, может принимать одно из двух значений:

`TN_EVENT_WCOND_OR` условие выполняется, если хотя бы один из битов маски флага совпадает с маской сравнения `wait_pattern`

`TN_EVENT_WCOND_AND` условие выполняется только в том случае, если хотя все биты маски флага совпадают с маской сравнения `wait_pattern`

`p_flags_pattern`

указатель по которому возвращается значение битовой маски флага если условие выполнено

Возвращаемые значения:

`TERR_WRONG_PARAM`

некорректное значение параметра ¹⁾

`TERR_NOEXS`

попытка обращения к объекту, не являющегося флагом ²⁾

`TERR_WCONTEXT`

попытка вызова сервиса из пользовательской критической секции или в прерывании

TERR_ILUSE

попытка проверки флага с атрибутом TN_EVENT_ATTR_SINGLE, если его очередь ожидания не пуста

TERR_TIMEOUT

условие не выполняется

TERR_NO_ERR

успешное выполнение

Пример вызова:

```
TN_EVENT event_test;
TN_UWORD flag;

if (tn_event_wait_polling(&event_test, 0x8000, TN_EVENT_WCOND_AND, &flag) == TERR_TIMEOUT)
{
    /* ... */
}
```

9.4. Сервисы управления очередями сообщений

9.4.1. Создание и удаление очереди сообщений

[tn_queue_create\(\)](#)

Функция предназначена для создания очереди сообщений. Поле `id_dque` очереди `dque` должно быть равно нулю до момента создания, таким образом уже созданные очереди защищаются от повторного создания.

Память для управляющей структуры `dque` и буфера сообщений должна быть выделена до момента создания очереди. Память может быть выделена на этапе компиляции (объявление глобальной переменной типа `TN_DQUE` для управляющей структуры и массива с элементами типа `*void` для буфера), либо динамически, если пользовательское приложение использует менеджер памяти. В последнем случае размер буфера сообщений должен быть равен (в байтах) `(sizeof(*void) * num_entries)`

Вызов:

```
TN_RETVAL tn_queue_create (TN_DQUE *dque,
                           void **data_fifo,
                           TN_UWORD num_entries
                           );
```

Разрешен вызов: В контексте задачи, в пользовательской критической секции, в прерывании

Параметры функции:

`dque` указатель на объект очереди сообщений. Структура `dque` типа `TN_DQUE` должна быть создана до момента вызова функции, статически или динамически

`data_fifo` указатель на буфер сообщений, который представляет собой массив элементов типа `*void`. Параметр может быть равен 0 или `TN_NULL` - в этом случае сервисы будут возвращать код ошибки `TERR_OUT_OF_MEM`.

`num_entries` размер буфера сообщений. Другими словами, максимальное количество сообщений, хранимых в очереди. Параметр должен быть равен количеству элементов в массиве `data_fifo`. Если указатель `data_fifo` равен 0 или `TN_NULL`, значение параметра может быть произвольным.

Внимание! Количество элементов очереди должно быть фактически на 1 больше, чем планируется использовать. Т.е. если `num_entries = 2`, то в очереди будет храниться один элемент и при попытке передачи второго сервис вернет ошибку. Не следует использовать очередь с одним элементом.

Возвращаемые значения:

`TERR_WRONG_PARAM` некорректное значение параметра ¹⁾
`TERR_EXS` попытка создания очереди, которая уже создана
`TERR_NO_ERR` успешное выполнение

Пример вызова:

```
#define QUEUE_SIZE 8
TN_DQUE queue      TN_DATA; /* объект типа очередь */
void *queue_fifo[QUEUE_SIZE] TN_DATA; /* буфер сообщений */
tn_queue_create(&queue, queue_fifo, QUEUE_SIZE);
```


tn_queue_delete()

Функция предназначена для удаления очереди сообщений. Поле `id_dque` очереди `dque` после выполнения сервиса устанавливается в 0.

Все задачи, ожидающие сообщения или ожидающие освобождения очереди будут переведены в состояние готовности к выполнению - сервисы приема и отсылки сообщения вернут код `TERR_DLT`.

Вызов:

```
TN_RETVAL tn_queue_delete (TN_DQUE *dque);
```

Разрешен вызов:

В контексте задачи

Параметры функции:

`dque`

указатель на очередь сообщений

Возвращаемые значения:

`TERR_WRONG_PARAM`

некорректное значение параметра ¹⁾

`TERR_NOEXS`

попытка удаления объекта, не являющегося очередью сообщений ²⁾

`TERR_WCONTEXT`

вызов функции из пользовательской критической секции или из обработчика прерывания

`TERR_NO_ERR`

успешное выполнение

Пример вызова:

```
TN_DQUE queue TN_DATA;
```

```
tn_queue_delete(&queue);
```

9.4.2. Отсылка сообщения

`tn_queue_send()`

Функция предназначена для отсылки сообщения `data_ptr` через очередь сообщений `dque`.

Если очередь задач, ожидающих сообщение не пуста, сервис переводит первую в очереди задачу в состояние готовой к выполнению и передает сообщение `data_ptr` в эту задачу, минуя буфер сообщений.

Если ни одна из задач системы не ожидает сообщение, параметр `data_ptr` кладется в конец буфера сообщений очереди и задача продолжает выполнение. Если буфер сообщений заполнен, то задача, посылающая сообщение переводится в состояние ожидания до тех пор, пока хотя бы одно сообщение из буфера не будет принято. При этом задача может выйти из ожидания с кодом `TERR_TIMEOUT` по истечении `timeout` системных тиков.

Вызов:

```
TN_RETVAL tn_queue_send (TN_DQUE  *dque,
                          void      *data_ptr,
                          TN_TIMEOUT timeout
                          );
```

Разрешен вызов: В контексте задачи

Параметры функции:

`dque` указатель на объект очереди сообщений.

`data_ptr` указатель на сообщение

`timeout` таймаут ожидания освобождения буфера в системных тиках. Значение параметра должно больше нуля. Если значение параметра равно `TN_WAIT_INFINITE`, задача, посылающая сообщение будет ожидать освобождения буфера.

Возвращаемые значения:

`TERR_WRONG_PARAM` некорректное значение параметра ¹⁾

`TERR_NOEXS` попытка обращения к объекту, который не является очередью ²⁾

`TERR_WCONTEXT` вызов сервиса в пользовательской критической секции или в обработке прерывания

`TERR_TIMEOUT` выход из сервиса по таймауту

`TERR_NO_ERR` успешное выполнение

Пример вызова:

```
TN_DQUE queue TN_DATA;
MY_MSG message;
message.a = 1;
message.b = 2;

tn_queue_send(&queue, (void*)&message, TN_WAIT_INFINITE);
```

Функция предназначена для отсылки сообщения `data_ptr` через очередь сообщений `dque` без блокирования вызывающей задачи.

Если очередь задач, ожидающих сообщение не пуста, сервис переводит первую в очереди задачу в состояние готовой к выполнению и передает сообщение `data_ptr` в эту задачу, минуя буфер сообщений.

Если ни одна из задач системы не ожидает сообщение, параметр `data_ptr` кладется в конец буфера сообщений очереди и задача продолжает выполнение. Если буфер сообщений заполнен, то функция возвращает код ошибки `TERR_TIMEOUT`.

Вызов:

```
TN_RETVAL tn_queue_send_polling (TN_DQUE *dque, void *data_ptr);
```

Разрешен вызов: В контексте задачи

Параметры функции:

`dque` указатель на объект очереди сообщений.

`data_ptr` указатель на сообщение

Возвращаемые значения:

`TERR_WRONG_PARAM` некорректное значение параметра ¹⁾

`TERR_NOEXS` попытка обращения к объекту, который не является очередью ²⁾

`TERR_WCONTEXT` вызов сервиса в пользовательской критической секции или в обработчике прерывания

`TERR_TIMEOUT` буфер сообщения заполнен, невозможно отослать сообщение

`TERR_NO_ERR` успешное выполнение

Пример вызова:

```
TN_DQUE queue TN_DATA;
MY_MSG message;

message.a = 1;
message.b = 2;

if (tn_queue_send_polling(&queue, (void*)&message) == TERR_NO_ERR)
{
    /* ... */
}
```

tn_queue_isend_polling()

Функция предназначена для отсылки сообщения `data_ptr` через очередь сообщений `dque` из прерывания.

Если очередь задач, ожидающих сообщение не пуста, сервис переводит первую в очереди задачу в состояние готовой к выполнению и передает сообщение `data_ptr` в эту задачу, минуя буфер сообщений.

Если ни одна из задач системы не ожидает сообщение, параметр `data_ptr` кладется в конец буфера сообщений очереди. Если буфер сообщений заполнен, то функция возвращает код ошибки `TERR_TIMEOUT`.

Вызов:

```
TN_RETVAL tn_queue_isend_polling (TN_DQUE *dque, void *data_ptr);
```

Разрешен вызов: В прерывании

Параметры функции:

`dque` указатель на объект очереди сообщений.

`data_ptr` указатель на сообщение

Возвращаемые значения:

<code>TERR_WRONG_PARAM</code>	некорректное значение параметра ¹⁾
<code>TERR_NOEXS</code>	попытка обращения к объекту, который не является очередью ²⁾
<code>TERR_WCONTEXT</code>	вызов сервиса в контексте задачи или в пользовательской критической секции
<code>TERR_TIMEOUT</code>	буфер сообщения заполнен, невозможно отослать сообщение
<code>TERR_NO_ERR</code>	успешное выполнение

Пример вызова:

```
extern TN_DQUE queue TN_DATA;
extern MY_MSG message;

tn_sys_interrupt (_T2Interrupt)    /* прерывание от таймера */
{
    IFS0bits.T2IF = 0;            /* сброс флага прерывания */

    message.a = 1;
    message.b = 2;

    if (tn_queue_isend_polling(&queue, (void*)&message) == TERR_NO_ERR)
    {
        /* ... */
    }
}
```

9.4.3. Прием сообщения

tn_queue_receive()

Функция предназначена для приема сообщения через очередь сообщений `dque`. Принятое сообщение (точнее говоря *адрес сообщения*) сохраняется по указателю `data_ptr`.

Если буфер очереди сообщений не пуст, функция передает первое сообщение в буфере по указателю `data_ptr`. Таким образом, после выхода из сервиса `data_ptr` будет указывать на сообщение, которое было отправлено через очередь. Если есть задача (задачи) ожидающая освобождения буфера для того чтобы отправить сообщение - эта задача будет переведена в состояние готовых к выполнению, а ее сообщение будет положено в очередь.

Если буфер очереди сообщений пуст, и нет задач, ожидающих освобождение буфера, то задача, вызвавшая сервис переводится в состояние ожидания. Если значение параметра `timeout` не равно `TN_WAIT_INFINITE`, то задача "проснется" по истечении `timeout` системных тиков с кодом ошибки `TERR_TIMEOUT`.

Вызов:

```
TN_RETVAL tn_queue_receive(TN_DQUE *dque,
                           void **data_ptr,
                           TN_TIMEOUT timeout
                           );
```

Разрешен вызов: В контексте задачи

Параметры функции:

`dque` указатель на объект очереди сообщений.

`data_ptr` указатель на указатель, в который сохраняется адрес сообщения

`timeout` время по истечении которого задача будет переведена из состояния ожидания в состояние готовых к выполнению, если не будет получено сообщения

Возвращаемые значения:

`TERR_WRONG_PARAM` некорректное значение параметра (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

`TERR_NOEXS` попытка обращения к объекту, который не является очередью (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

`TERR_WCONTEXT` вызов сервиса в пользовательской критической секции или в обработчике прерывания

`TERR_TIMEOUT` выход из функции по таймауту

`TERR_NO_ERR` успешное выполнение

Пример вызова:

```
TN_DQUE queue TN_DATA; /* очередь сообщений */
MY_MSG *message; /* указатель на сообщение */

tn_queue_receive(&queue, &message, 10);
if (message->a)
{
    /* ... */
}
```

Функция предназначена для приема сообщения через очередь сообщений `dque`. Принятое сообщение (точнее говоря *адрес сообщения*) сохраняется по указателю `data_ptr`. Если в очереди отсутствуют сообщения, то задача *не блокируется*, а функция возвращает код ошибки `TERR_TIMEOUT`.

Если буфер очереди сообщений не пуст, функция передает первое сообщение в буфере по указателю `data_ptr`. Таким образом, после выхода из сервиса `data_ptr` будет указывать на сообщение, которое было отправлено через очередь. Если есть задача (задачи) ожидающая освобождения буфера для того чтобы отправить сообщение - эта задача будет переведена в состояние готовых к выполнению, а ее сообщение будет положено в очередь.

Вызов:

<code>TN_RETVAL tn_queue_receive_polling (TN_DQUE *dque, void **data_ptr);</code>

Разрешен вызов: В контексте задачи

Параметры функции:

`dque` указатель на объект очереди сообщений.

`data_ptr` указатель на указатель, в который сохраняется адрес сообщения

Возвращаемые значения:

`TERR_WRONG_PARAM` некорректное значение параметра (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

`TERR_NOEXS` попытка обращения к объекту, который не является очередью (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

`TERR_WCONTEXT` вызов сервиса в пользовательской критической секции или в обработчике прерывания

`TERR_TIMEOUT` очередь сообщений пуста

`TERR_NO_ERR` успешное выполнение

Пример вызова:

<pre>TN_DQUE queue TN_DATA; /* очередь сообщений */ MY_MSG *message; /* указатель на сообщение */ if (tn_queue_receive_polling(&queue, &message) == TERR_NO_ERR) { if (message->a) { /* ... */ } }</pre>
--

Функция предназначена для приема сообщения через очередь сообщений `dque` **в прерывании**. Принятое сообщение (точнее говоря *адрес сообщения*) сохраняется по указателю `data_ptr`.

Если буфер очереди сообщений не пуст, функция передает первое сообщение в буфере по указателю `data_ptr`. Таким образом, после выхода из сервиса `data_ptr` будет указывать на сообщение, которое было отправлено через очередь. Если есть задача (задачи) ожидающая освобождения буфера для того чтобы отправить сообщение - эта задача будет переведена в состояние готовых к выполнению, а ее сообщение будет положено в очередь.

Если буфер очереди сообщений пуст, функция возвращает код ошибки `TERR_TIMEOUT`.

Вызов:

```
TN_RETVAL tn_queue_ireceive (TN_DQUE *dque, void **data_ptr);
```

Разрешен вызов: В обработке прерывания

Параметры функции:

`dque` указатель на объект очереди сообщений.

`data_ptr` указатель на указатель, в который сохраняется адрес сообщения

Возвращаемые значения:

`TERR_WRONG_PARAM` некорректное значение параметра (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

`TERR_NOEXS` попытка обращения к объекту, который не является очередью (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

`TERR_WCONTEXT` вызов сервиса в пользовательской критической секции или в контексте задачи

`TERR_TIMEOUT` очередь сообщений пуста

`TERR_NO_ERR` успешное выполнение

Пример вызова:

```
extern TN_DQUE queue TN_DATA;
tn_sys_interrupt (_T2Interrupt) /* прерывание от таймера */
{
    MY_MSG message;
    IFS0bits.T2IF = 0;          /* сброс флага прерывания */

    if (tn_queue_ireceive(&queue, &message) == TERR_NO_ERR)
    {
        if (message->a)
        {
            /* ... */
        }
    }
}
```

9.5. Сервисы управления мьютексами

9.5.1. Создание и удаление мьютекса

[tn_mutex_create\(\)](#)

Функция предназначена для создания мьютекса. Поле `id_mutex` структуры `mutex` должно быть равно нулю до момента создания мьютекса. Таким образом уже созданные мьютексы защищаются от повторного создания.

Память для управляющей структуры `mutex` должна быть выделена до момента создания мьютекса. Память может быть выделена на этапе компиляции (объявление глобальной переменной типа `TN_MUTEX`), либо динамически, если пользовательское приложение использует менеджер памяти.

Параметр `attribute` указывает тип протокола обхода инверсии приоритетов:

`TN_MUTEX_ATTR_CEILING` - если используется протокол увеличения приоритета или

`TN_MUTEX_ATTR_INHERIT`, если используется протокол наследования приоритета.

Если параметр `attribute` равен `TN_MUTEX_ATTR_CEILING`, необходимо указать параметр `ceil_priority` - максимальный приоритет из задач, который могут владеть мьютексом. Если же используется протокол наследования приоритета, то параметр `ceil_priority` игнорируется.

Вызов:

```
TN_RETVAL tn_mutex_create(TN_MUTEX *mutex, TN_UWORD attribute, TN_UWORD ceil_priority);
```

Разрешен вызов: В контексте задачи, в прерывании, в пользовательской критической секции

Параметры функции:

`mutex` указатель на структуру мьютекса типа `TN_MUTEX`. Структура должна быть создана до момента вызова функции, статически или динамически

`attribute` тип протокола обхода инверсии приоритетов, используемый мьютексом. Параметр может принимать одно из двух значений:

`TN_MUTEX_ATTR_CEILING` Используется протокол увеличения приоритета

`TN_MUTEX_ATTR_INHERIT` Используется протокол наследования приоритета

`ceil_priority` максимальный приоритет из всех задач, которые могут владеть мьютексом. Допустим, мьютексом могут владеть задачи с приоритетом 3, 4, 9 и 2. Максимальный приоритет - 2. Параметр игнорируется, если `attribute = TN_MUTEX_ATTR_INHERIT`

Возвращаемые значения:

`TERR_WRONG_PARAM` некорректное значение параметра (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

`TERR_EXS` попытка создания мьютекса, который уже создан

`TERR_NO_ERR` успешное выполнение

Пример вызова:

```
TN_MUTEX mutex_test;

/* Создается мьютекс с протоколом увеличения приоритета.
   Максимальный приоритет из задач, которые используют мьютекс равен 2
*/

tn_mutex_create(&mutex_test, TN_MUTEX_ATTR_CEILING, 2);
```


tn_mutex_delete()

Функция предназначена для удаления мютекса. Поле `id_mutex` структуры `mutex` после выполнения сервиса устанавливается в 0. Приоритет задачи, владеющий мютексом будет восстановлен, если он был изменен вследствие протокола обхода инверсии приоритетов.

Задача владеющая мютексом никак не будет извещена о его удалении, однако, сервис освобождения мютекса вернет код ошибки `TERR_DLT`. Все задачи, ожидающие освобождения мютекса будут переведены в состояние `RUNNABLE`.

Вызов:

```
TN_RETURN tn_mutex_delete(TN_MUTEX *mutex);
```

Разрешен вызов:

В контексте задачи

Параметры функции:

`mutex`

указатель на структуру удаляемого мютекса

Возвращаемые значения:

`TERR_WRONG_PARAM`

некорректное значение параметра (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

`TERR_WCONTEXT`

вызов сервиса в обработчике системного прерывания или в пользовательской критической секции

`TERR_NOEXS`

попытка удаления несуществующего мютекса

`TERR_NO_ERR`

успешное выполнение

Пример вызова:

```
extern TN_MUTEX mutex_test;  
  
tn_mutex_delete(&mutex_test);
```

9.5.2. Блокировка мьютекса

`tn_mutex_lock()`

Функция предназначена для блокировки мьютекса. Если мьютекс еще не заблокирован, он блокируется и задача, которая вызвала сервис не переводится в состояние ожидания. Если мьютекс заблокирован, задача переводится в состояние `WAITING` и ставится в очередь задач, ожидающих освобождение мьютекса.

Параметр `timeout` задает время ожидания мьютекса в системных тиках. Если в течении этого времени мьютекс не будет захвачен задачей, она удаляется из очереди ожидающих и переводится в состояние готовых к выполнению. Если значение параметра `timeout` равно `TN_WAIT_INFINITE`, то задача будет ожидать мьютекс до тех пор, пока он не освободиться.

Если задача заблокировала мьютекс ранее, сервис возвращает код ошибки `TERR_ILUSE`. Так же этот код возвращается в том случае, если задача пытается заблокировать мьютекс с протоколом увеличения приоритета (Priority Ceiling Protocol) и ее приоритет выше порога, заданного при создании мьютекса.

Вызов:

```
TN_RETURN tn_mutex_lock(TN_MUTEX *mutex, TN_TIMEOUT timeout);
```

Разрешен вызов: В контексте задачи

Параметры функции:

`mutex` указатель на структуру блокируемого мьютекса
`timeout` таймаут в течении которого задача ожидает освобождения мьютекса

Возвращаемые значения:

`TERR_WRONG_PARAM` некорректное значение параметра (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

`TERR_NOEXS` попытка блокировки несуществующего мьютекса

`TERR_WCONTEXT` попытка блокировки мьютекса из обработчика прерывания или пользовательской критической секции

`TERR_ILUSE`

мьютекс уже заблокирован задачей, вызвавшей сервис

мьютекс использует протокол увеличения приоритета, а задача, пытающаяся заблокировать его имеет приоритет выше порогового

`TERR_TIMEOUT` выход из сервиса по таймауту

`TERR_NO_ERR` успешное выполнение

Пример вызова:

```
TN_MUTEX mutex_test;  
  
tn_mutex_lock(&mutex_test, 10);
```

Функция предназначена для захвата мьютекса без блокировки задачи. Если мьютекс еще не заблокирован, он блокируется. Если мьютекс заблокирован, сервис возвращает код ошибки `TERR_TIMEOUT`.

Если задача заблокировала мьютекс ранее, сервис возвращает код ошибки `TERR_ILUSE`. Так же этот код возвращается в том случае, если задача пытается заблокировать мьютекс с протоколом увеличения приоритета (Priority Ceiling Protocol) и ее приоритет выше порога, заданного при создании мьютекса.

Вызов:

```
TN_RETVAL tn_mutex_lock_polling(TN_MUTEX *mutex);
```

Разрешен вызов: В контексте задачи

Параметры функции:

`mutex` указатель на структуру блокируемого мьютекса

Возвращаемые значения:

`TERR_WRONG_PARAM` некорректное значение параметра (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

`TERR_NOEXS` попытка блокировки несуществующего мьютекса

`TERR_WCONTEXT` попытка блокировки мьютекса из обработчика прерывания или пользовательской критической секции

`TERR_ILUSE`

мьютекс уже заблокирован задачей, вызвавшей сервис

мьютекс использует протокол увеличения приоритета, а задача, пытающаяся заблокировать его имеет приоритет выше порогового

`TERR_TIMEOUT` мьютекс уже заблокирован другой задачей

`TERR_NO_ERR` успешное выполнение

Пример вызова:

```
TN_MUTEX mutex_test;

if (tn_mutex_lock_polling(&mutex_test) == TERR_NO_ERR)
{
    /* успешная попытка блокировки мьютекса */
}
```

9.5.3. Освобождение мютекса

tn_mutex_unlock()

Функция предназначена для освобождения мютекса. Если есть задачи, ожидающие мютекс, то задача, стоящая первая в очереди, переводится в состояние готовой к выполнению, а ее приоритет может измениться согласно протоколу обхода инверсии приоритетов.

Если задача, вызвавшая сервис, пытается разблокировать свободный мютекс, сервис возвращает код ошибки `TERR_ILUSE`.

Текущая задача после освобождения мютекса может изменить приоритет согласно протоколу обхода инверсии приоритетов.

Вызов:

`TN_RETVAL tn_mutex_unlock(TN_MUTEX *mutex);`

Разрешен вызов: В контексте задачи

Параметры функции:

`mutex`

указатель на структуру освобождаемого мютекса

Возвращаемые значения:

`TERR_WRONG_PARAM` некорректное значение параметра (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

`TERR_NOEXS` попытка освобождения несуществующего мютекса

`TERR_WCONTEXT` попытка освобождения мютекса из обработчика прерывания или пользовательской критической секции

`TERR_ILUSE` мютекс уже освобожден

`TERR_NO_ERR` успешное выполнение

Пример вызова:

`TN_MUTEX mutex_test;
tn_mutex_unlock(&mutex_test);`

9.6. Сервисы управления пулами блоков памяти

9.6.1. Создание и удаление пула

[tn_fmem_create\(\)](#)

Функция предназначена для создания пула блоков памяти фиксированного размера. Поле `id_fmp` структуры `fmp` должно быть равно нулю до момента создания пула. Таким образом уже созданные пулы защищаются от повторного создания.

Память для самого пула так же должна быть выделена до вызова сервиса. Память может быть выделена статически или динамически, указатель на память передается в качестве параметра `start_addr`.

Для лучшего использования памяти, желательно чтобы размер блока (`block_size`) был кратным машинному слову: для PIC24/dsPIC это 2, 4, 6 байт. Для удобного выделения памяти (автоматического выравнивания) можно использовать макрос `MAKE_ALIG`:

```
TN_FMP my_pool;
TN_UWORD my_pool_mem[NUM_BLOCKS * (MAKE_ALIG(BLOCK_SIZE) / sizeof(TN_UWORD))];

tn_fmem_create (&my_pool, my_pool_mem, BLOCK_SIZE, NUM_BLOCKS);
```

Вызов:

```
TN_RETVAL tn_fmem_create (TN_FMP *fmp,
                          void *start_addr,
                          TN_UWORD block_size,
                          TN_UWORD num_blocks
                          );
```

Разрешен вызов: В контексте задачи, в обработчике прерывания, в пользовательской критической секции

Параметры функции:

`fmp` указатель на структуру пула типа `TN_FMP`. Структура должна быть создана до момента вызова функции, статически или динамически

`start_addr` указатель на память, выделенную для пула. Размер выделенной памяти должен быть не меньше `block_size * num_blocks` байт

`block_size` размер блока памяти в байтах

`num_blocks` количество блоков памяти в пуле

Возвращаемые значения:

`TERR_WRONG_PARAM` некорректное значение параметра ¹⁾

`TERR_EXS` попытка создания пула, который уже создан

`TERR_NO_ERR` успешное выполнение

Пример вызова:

```
TN_FMP my_pool;
TN_UWORD my_pool_mem[NUM_BLOCKS * (MAKE_ALIG(BLOCK_SIZE) / sizeof(TN_UWORD))];

tn_fmem_create (&my_pool, my_pool_mem, BLOCK_SIZE, NUM_BLOCKS);
```

tn_fmem_delete()

Функция предназначена для удаления пула блоков памяти фиксированного размера. Поле `id_fmp` структуры `fmp` после выполнения сервиса устанавливается в 0.

Все задачи, ожидающие освобождения блока выходят из сервиса запроса с кодом ошибки `TERR_DLT`.

Вызов:

```
TN_RETVAL tn_fmem_delete (TN_FMP *fmp);
```

Разрешен вызов:

В контексте задачи

Параметры функции:

`fmp`

указатель на структуру удаляемого пула

Возвращаемые значения:

`TERR_WRONG_PARAM`

некорректное значение параметра ¹⁾

`TERR_NOEXS`

попытка удаления несуществующего пула

`TERR_WCONTEXT`

попытка вызова сервиса из обработчика прерывания или из пользовательской критической секции

`TERR_NO_ERR`

успешное выполнение

Пример вызова:

```
TN_FMP my_pool;
```

```
tn_fmem_delete(&my_pool);
```

9.6.2. Получение блока памяти

tn_fmem_get()

Функция предназначена для получения блока памяти из пула. Указатель на получаемый блок передается в сервис в качестве параметра `p_data`. Содержимое блока памяти после получения его задачей не определено.

Если в пуле есть свободные блоки, то один из них выделяется для задачи, и сервис возвращает код `TERR_NO_ERR`. Если свободные блоки в пуле отсутствуют, задача переводится в состояние ожидания и ставится в конец очереди задач, ожидающих освобождения блока памяти.

Параметр `timeout` задает время ожидания блока памяти в системных тиках. Если в течении этого времени блок не будет выделен для задачи, она удаляется из очереди ожидающих блок и переводится в состояние готовых к выполнению. Сервис возвращает значение `TERR_TIMEOUT`.

Вызов:

```
TN_RETVAL tn_fmem_get (TN_FMP *fmp, void **p_data, TN_TIMEOUT timeout);
```

Разрешен вызов: В контексте задачи

Параметры функции:

<code>fmp</code>	казатель на пул из которого будет выделяться блок памяти
<code>p_data</code>	указатель на указатель, который после успешного выполнения сервиса будет содержать адрес выделенного блока
<code>timeout</code>	таймаут в системных тиках, в течении которого задача будет ожидать освобождения блока

Возвращаемые значения:

<code>TERR_WRONG_PARAM</code>	некорректное значение параметра ¹⁾
<code>TERR_NOEXS</code>	попытка получения блока из несуществующего пула
<code>TERR_WCONTEXT</code> секции	вызов сервиса из обработчика прерывания или пользовательской критической секции
<code>TERR_TIMEOUT</code>	выход из сервиса по таймауту
<code>TERR_NO_ERR</code>	успешное выполнение

Пример вызова:

```
#define BLOCK_SIZE 4

TN_FMP my_pool;
TN_UWORD my_pool_mem[NUM_BLOCKS * (MAKE_ALIG(BLOCK_SIZE) / sizeof(TN_UWORD))];

TN_UWORD *block_pt;

tn_fmem_create (&my_pool, my_pool_mem, BLOCK_SIZE, NUM_BLOCKS);
tn_fmem_get (&my_pool, &block_pt, TN_WAIT_INFINITE);
```

[tn_fmем_get_polling\(\)](#)

Функция предназначена для получения блока памяти из пула без блокировки задачи. Указатель на получаемый блок передается в сервис в качестве параметра `p_data`. Содержимое блока памяти после получения его задачей не определено.

Если в пуле есть свободные блоки, то один из них выделяется для задачи, и сервис возвращает код `TERR_NO_ERR`. Если свободные блоки в пуле отсутствуют, сервис возвращает код ошибки `TERR_TIMEOUT`.

Вызов:

`TN_RETURN tn_fmем_get_polling (TN_FMP *fmp, void **p_data);`

Разрешен вызов: В контексте задачи

Параметры функции:

`fmp` указатель на пул из которого будет выделяться блок памяти

`p_data` указатель на указатель, который после успешного выполнения сервиса будет содержать адрес выделенного блока

Возвращаемые значения:

`TERR_WRONG_PARAM` некорректное значение параметра (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

`TERR_NOEXS` попытка получения блока из несуществующего пула

`TERR_WCONTEXT` вызов сервиса из обработчика прерывания или пользовательской критической секции

`TERR_TIMEOUT` в пуле отсутствуют свободные блоки памяти

`TERR_NO_ERR` успешное выполнение

Пример вызова:

```
#define BLOCK_SIZE 4

TN_FMP my_pool;
TN_UWORD my_pool_mem[NUM_BLOCKS * (MAKE_ALIG(BLOCK_SIZE) / sizeof(TN_UWORD))];

TN_UWORD *block_pt;

tn_fmем_create(&my_pool, my_pool_mem, BLOCK_SIZE, NUM_BLOCKS);
if (tn_fmем_get(&my_pool, &block_pt) == TERR_TIMEOUT)
{
    /* в пуле нет свободных блоков памяти */
}
```


tn_fmem_get_ipolling()

Функция предназначена для получения блока памяти из пула в прерывании. Указатель на получаемый блок передается в сервис в качестве параметра `p_data`. Содержимое блока памяти после получения его задачей не определено.

Если в пуле есть свободные блоки, то один из них выделяется для задачи, и сервис возвращает код `TERR_NO_ERR`. Если свободные блоки в пуле отсутствуют, сервис возвращает код ошибки `TERR_TIMEOUT`.

Вызов:

```
TN_RETURN tn_fmem_get_ipolling (TN_FMP *fmp, void **p_data);
```

Разрешен вызов: В прерывании

Параметры функции:

`fmp` указатель на пул из которого будет выделяться блок памяти

`p_data` указатель на указатель, который после успешного выполнения сервиса будет содержать адрес выделенного блока

Возвращаемые значения:

`TERR_WRONG_PARAM` некорректное значение параметра (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

`TERR_NOEXS` попытка получения блока из несуществующего пула

`TERR_WCONTEXT` вызов сервиса из контекста задачи или из пользовательской критической секции

`TERR_TIMEOUT` в пуле отсутствуют свободные блоки памяти

`TERR_NO_ERR` успешное выполнение

Пример вызова:

```
extern TN_FMP my_pool;

tn_sys_interrupt (_T2Interrupt)    /* прерывание от таймера */
{
    TN_UWORD *block;

    IFS0bits.T2IF = 0;               /* сброс флага прерывания */
    if (tn_fmem_get_ipolling(&my_pool, &block) == TERR_NO_ERR)
    {
        /* ... */
    }
    /* ... */
}
```

9.6.3. Освобождение блока памяти

tn_fmem_release()

Функция предназначена для освобождения выделенного блока памяти. Указатель на освобождаемый блок передается в сервис в качестве параметра `p_data`. Функция не проверяет принадлежность блока `p_data` к пулу `fmp`.

Если в очереди ожидания блоков памяти есть задачи, то первая задача из очереди будет переведена в состояние готовности к выполнению.

Вызов:

```
TN_RETURN tn_fmem_release(TN_FMP *fmp, void *p_data);
```

Разрешен вызов: В контексте задачи

Параметры функции:

`fmp` указатель на пул, блок которого освобождается задачей
`p_data` указатель на блок, который будет освобождаться задачей

Возвращаемые значения:

`TERR_WRONG_PARAM` некорректное значение параметра (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

`TERR_NOEXS` попытка освобождения блока несуществующего пула

`TERR_WCONTEXT` вызов сервиса из обработчика прерывания или пользовательской критической секции

`TERR_NO_ERR` успешное выполнение

Пример вызова:

```
#define BLOCK_SIZE 4

TN_FMP my_pool;
TN_UWORD my_pool_mem[NUM_BLOCKS * (MAKE_ALIG(BLOCK_SIZE) / sizeof(TN_UWORD))];

TN_UWORD *block_pt;

tn_fmem_create (&my_pool, my_pool_mem, BLOCK_SIZE, NUM_BLOCKS);
tn_fmem_get (&my_pool, &block_pt, TN_WAIT_INFINITE);

/* ... */

tn_fmem_release(&my_pool, block_pt);
```

tn_fmem_irelease()

Функция предназначена для освобождения выделенного блока памяти в прерывании. Указатель на освобождаемый блок передается в сервис в качестве параметра `p_data`. Функция не проверяет принадлежность блока `p_data` к пулу `fmp`.

Если в очереди ожидания блоков памяти есть задачи, то первая задача из очереди будет переведена в состояние готовности к выполнению.

Вызов:

```
TN_RETVAL tn_fmem_irelease(TN_FMP *fmp, void *p_data);
```

Разрешен вызов: В прерывании

Параметры функции:

`fmp` указатель на пул, блок которого освобождается задачей

`p_data` указатель на блок, который будет освобождаться задачей

Возвращаемые значения:

`TERR_WRONG_PARAM` некорректное значение параметра (*замечание:* данный код возврата возможен только в случае использования сервисов с проверкой параметров)

`TERR_NOEXS` попытка освобождения блока несуществующего пула

`TERR_WCONTEXT` вызов сервиса в контексте задачи или из пользовательской критической секции

`TERR_NO_ERR` успешное выполнение

Пример вызова:

```
extern TN_FMP my_pool;

tn_sys_interrupt (_T2Interrupt)    /* прерывание от таймера */
{
    TN_UWORD *block_pt;

    IFS0bits.T2IF = 0;              /* сброс флага прерывания */

    if (tn_fmem_get_ipolling(&my_pool, &block) == TERR_NO_ERR)
    {
        /* ... */
        tn_fmem_irelease(&my_pool, block_pt);
    }
}
```

9.7. Системные сервисы

9.7.1. Основные сервисы

[tn_start_system\(\)](#)

Функция предназначена для запуска системы. В функции создаются две системные задачи - задача таймера (timer) и задача простоя (idle), затем производится переключение контекста на задачу таймера. Возврат из функции не производится.

Вызов:

```
void tn_start_system (TN_UWORD *timer_task_stack,
                     TN_UWORD timer_task_stack_size,
                     TN_UWORD *idle_task_stack,
                     TN_UWORD idle_task_stack_size,
                     void (*app_in_cb)(void),
                     void (*cpu_int_en)(void),
                     void (*idle_user_cb)(void)
                     );
```

Разрешен вызов:

Один раз в функции main()

Параметры функции:

timer_task_stack

указатель на стек задачи системного таймера

timer_task_stack_size

размер стека задачи системного таймера

idle_task_stack

указатель на стек системной задачи простоя

idle_task_stack_size

размер стека системной задачи простоя

app_in_cb

указатель на функцию инициализации приложения. Функция вызывается один раз при старте системы

cpu_int_en

указатель на функцию конфигурации и разрешения прерываний. Функция вызывается один раз при старте системы после вызова функции app_in_cb

idle_user_cb

указатель на функцию, которая периодически вызывается из системной задачи простоя

Возвращаемые значения:

нет

Пример вызова:

```
#define TMR_TASK_STACK_SIZE 128
#define IDL_TASK_STACK_SIZE 64

TN_UWORD stk_tmr_task[TMR_TASK_STACK_SIZE];
TN_UWORD stk_idl_task[IDL_TASK_STACK_SIZE];

void Appl_Init (void)
{
    /* инициализация приложения */
    /* инициализация системного таймера */
}
```

```

    /* создание пользовательских задач */
}

void Int_Init (void)
{
    /* разрешение прерывания от системного таймера */
    /* разрешение других прерываний */
}

void IDLE_func (void)
{
    /* */
}

int main (void)
{
    tn_start_system (stk_tmr_task,
                    TMR_TASK_STACK_SIZE,
                    stk_idl_task,
                    IDL_TASK_STACK_SIZE,
                    Appl_Init,
                    Int_Init,
                    IDLE_func
                    );
    return (0);
}

```

tn_tick_int_processing()

Функция предназначена для обслуживания системного таймера. Вызов функции должен производиться строго в прерывании. Как правило это прерывание от некоего периодического источника - таймера ядра или периферийного таймера.

Вызов:

```
void tn_tick_int_processing (void);
```

Разрешен вызов:

В прерывании

Параметры функции:

нет

Возвращаемые значения:

нет

Пример вызова:

```
tn_sys_interrupt (_T2Interrupt)    /* прерывание от таймера TMR2 */
{
    IFS0bits.T2IF = 0;             /* сброс флага прерывания */
    tn_tick_int_processing();
}
```

tn_sys_tslice_ticks()

Функция устанавливает период переключения задач по карусельному методу (round-robin) для выбранного приоритета. Возможна установка индивидуального периода переключения для каждого приоритета.

Вызов:

TN_RETVAL tn_sys_tslice_ticks (TN_UWORD priority, TN_UWORD value);
--

Разрешен вызов:

В контексте задачи

Параметры функции:

priority

приоритет задач для которого устанавливается период переключения по методу round-robin.

priority $\in [1 \dots (\text{TN_NUM_PRIORITY} - 2)]$

value

величина кванта времени, выделяемого каждой задаче (период переключения) в системных тиках.

value $\in [\text{NO_TIME_SLICE} \dots \text{MAX_TIME_SLICE}]$, где $\text{NO_TIME_SLICE} = 0$, а $\text{MAX_TIME_SLICE} = (\text{UINT_MAX} - 1)$. Если value = NO_TIME_SLICE, карусельное планирование для задач с приоритетом priority не осуществляется.

Возвращаемые значения:

TERR_WRONG_PARAM

некорректное значение параметра ¹⁾

TERR_WCONTEXT

попытка вызова функции в прерывании или в пользовательской критической секции

TERR_NO_ERR

успешное выполнение

Пример вызова:

<pre>/* установка round-robin кванта равного 10 системных тиков для задач с приоритетом равным 10 */ tn_sys_tslice_ticks(10, 10); /* запрещение карусельного планирования для задач с приоритетом равным 10 */ tn_sys_tslice_ticks(10, NO_TIME_SLICE);</pre>

tn_sys_context_get()

Функция возвращает текущий контекст системы. Сервис можно использовать для проверки текущего контекста в функции, которая может вызываться как из контекста задачи, так и из пользовательского прерывания.

Вызов:

```
TN_CONTEXT tn_sys_context_get (void);
```

Разрешен вызов:

В контексте задачи, в прерывании, в пользовательской критической секции

Параметры функции:

нет

Возвращаемые значения:

TN_CONTEXT_TASK

Контекст задачи

TN_CONTEXT_SYS_INT

Системное прерывание

TN_CONTEXT_CRITICAL

Пользовательская критическая секция

Пример вызова:

```
void foo (void)
{
    TN_CONTEXT context;

    context = tn_sys_context_get();
    if (context == TN_CONTEXT_TASK)
    {
        tn_sem_signal(&sem);
    }
    else if (context == TN_CONTEXT_SYS_INT)
    {
        tn_sem_isignal(&sem);
    }
}
```


9.7.2. Запрещение переключения контекста

[tn_sys_enter_critical\(\)](#)

Вызов функции запрещает переключение контекста (в том числе и системные прерывания) до тех пор, пока не будет вызвана парная функция [tn_sys_exit_critical\(\)](#).

Вызов:

```
void tn_sys_enter_critical (void);
```

Разрешен вызов:

В контексте задачи

Параметры функции:

нет

Возвращаемые значения:

нет

Пример вызова:

```
long long Global_Variable;

void TN_TASK Task_1 (void *param)
{
    for (;;)
    {
        tn_sys_enter_critical();
        Global_Variable = 1255456;
        tn_sys_exit_critical();
    }
}

void TN_TASK Task_2 (void *param)
{
    long long tmp;

    for (;;)
    {
        tn_sem_acquire(&sem_rc, TN_WAIT_INFINITE);

        tn_sys_enter_critical();
        tmp = Global_Variable;
        tn_sys_exit_critical();

        if (tmp)
            tn_sem_signal(&sem_tr);
    }
}
```

[tn_sys_exit_critical\(\)](#)

Вызов функции разрешает переключение контекста и системные прерывания после вызова парной функции [tn_sys_enter_critical\(\)](#).

Вызов:

```
void tn_sys_exit_critical (void);
```

Разрешен вызов:

В контексте задачи

Параметры функции:

нет

Возвращаемые значения:

нет

Пример вызова:

```
long long Global_Variable;

void TN_TASK Task_1 (void *param)
{
    for (;;)
    {
        tn_sys_enter_critical();
        Global_Variable = 1255456;
        tn_sys_exit_critical();
    }
}

void TN_TASK Task_2 (void *param)
{
    long long tmp;

    for (;;)
    {
        tn_sem_acquire(&sem_rc, TN_WAIT_INFINITE);

        tn_sys_enter_critical();
        tmp = Global_Variable;
        tn_sys_exit_critical();

        if (tmp)
            tn_sem_signal(&sem_tr);
    }
}
```

9.7.3. Системное время

tn_sys_time_get()

Функция возвращает значение системных часов - 32-битной беззнаковой переменной, инкрементируемой каждый системный тик.

Вызов:

```
TN_SYS_TIM_T tn_sys_time_get (void);
```

Разрешен вызов:

В контексте задачи, в системном прерывании

Параметры функции:

нет

Возвращаемые значения:

TN_SYS_TIM_T

значение системных часов в системных тиках

Пример вызова:

```
void TN_TASK Task (void *param)
{
    TN_SYS_TIM_T sys_time;

    for (;;)
    {
        sys_time = tn_sys_time_get();
        foo();
        sys_time = tn_sys_time_get() - sys_time;

        if (sys_time < TASK_CALL_PERIOD)
            tn_task_sleep(TASK_CALL_PERIOD - sys_time);
        else
            tn_task_sleep(1);
    }
}
```

tn_sys_time_set()

Функция устанавливает системные часы - 32-битной беззнаковую переменную, инкрементируемую каждый системный тик.

Вызов:

```
void tn_sys_time_get (TN_SYS_TIM_T value);
```

Разрешен вызов:

В контексте задачи, в системном прерывании

Параметры функции:

value

величина присваиваемая системным часам (в системных тиках)

Возвращаемые значения:

нет

Пример вызова:

```
void TN_TASK Task (void *param)
{
    TN_SYS_TIM_T sys_time;

    for (;;)
    {
        tn_sys_time_set(0);
        foo();
        sys_time = tn_sys_time_get();

        if (sys_time < TASK_CALL_PERIOD)
            tn_task_sleep(TASK_CALL_PERIOD - sys_time);
        else
            tn_task_sleep(1);
    }
}
```