MULTICLET®

www.multiclet.com

РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ.
ОС FReeRTOS для мультиклеточного процессора MultiClet P1

Содержание

1 Общие сведения

FreeRTOS – многозадачная операционная система реального времени (ОСРВ) для встраиваемых систем. Распространяется под модифицированной лицензией GPL с исключением, позволяющим разработчику присвоить модифицированный код операционной системы.

Основные особенности ОС:

- FreeRTOS фактически можно считать ОС жесткого реального времени, но это будет зависеть от приложения, в котором она используется.
- Планировщик может быть вытесняющим или кооперативным. Все зависит от конфигурации ОС. Кооперативный планировщик не имеет приоритетов и просто передает управление от одной задачи к другой. Вытесняющий планировщик выбирает задачу с максимальным на данный момент приоритетом, его работа основана на приоритетах, которые присваиваются при создании задачам.
- Имеется возможность использования сопрограмм (co-routine).
- Динамическое планирование осуществляется через равные промежутки времени равные тикам системного таймера. Частота тиков задается на этапе разработки приложения.
- Алгоритм планирования основан на выборе самого большого на данный момент приоритета. Если имеется больше чем одна задача с самым высоким приоритетом, то выполнение происходит по очереди.
- Межпроцессорное взаимодействие:
  - Очередь – большая часть информации передается через ссылки для экономии памяти. Очередь может читать или писать внутри обработчиков прерываний не блокируясь. Все очереди могут читать или писать блочно с настраиваемыми задержками.
  - Синхронизация – можно использовать бинарные, счетные и рекурсивные семафоры, а также мьютексы. Они используются атомарно, т. е. когда семафор берется или отдается прерывания запрещены, планировщик временно приостанавливает свою работу.
- Таймаут на каждом блоке уменьшает вероятность тупика для доступа к ресурсам.
- Имеются критические секции, в которых запрещены прерывания. Критические секции в пределах задачи могут быть вложенными. И каждая задача отслеживает количество вложений. Имеется возможность уступить процессор внутри критической секции (при использовании кооперативного планировщика).
- Приостановка планировщика может использоваться для получения полного доступа к микроконтроллеру.

- Во FreeRTOS имеется две модели выделения памяти. Первая модель предусматривает простое выделение памяти при создании каждой задачи, но не имеет механизма освобождения или повторного использования памяти (поэтому задачи не могут быть удалены). Вторая модель позволяет выделять и распределять память и использует самый пригодный для распределения памяти алгоритм, однако не может комбинировать смежные свободные части памяти.
- FreeRTOS не имеет никаких способов борьбы с инверсией приоритетов.

Для работы любой ОС необходим эталон времени. Этот эталон используется для генерации периодических прерываний, которые позволяют прерывать работу задачи и давать ядру возможность периодически выполнять свою работу. В данной ОС этим эталонным интервалом времени называется тик (Tick), который используется в качестве базовой единицы измерения времени в системе.
Основу приложения составляет ядро системы, обеспечивающее необходимую функциональность, и прикладные задачи пользователя.

Запуск приложения осуществляется в два этапа:

- Настройка приложения, создание задач пользователя.
- Запуск планировщика задач.

Вся основная функциональность системы заключается в планировщике задач. Он осуществляет распределение процессорного времени между задачами, выбирает задачу, которая должна запуститься сейчас. Планировщик задач в случае необходимости сохраняет контекст текущей активной задачи и восстанавливает контекст задачи, назначенной к исполнению.
Контекст задачи – это набор данных, содержащий всю необходимую информацию для возобновления выполнения задачи с того места, где она была ранее прервана - обычно содержит значение рабочих регистров микроконтроллера, счетчика команд, слово состояния. Такое переключение контекстов является основным механизмом работы ОС при переходе от выполнения одной задачи к выполнению другой.
Запуск планировщика осуществляется вызовом vTaskStartScheduler. ***До запуска планировщика задачи не переключаются и не выполняются!*** Для остановки планировщика используется вызов vTaskEndScheduler, который осуществляет остановку всех задач, освобождение памяти и далее программа продолжает работу с места следующего за вызовом vTaskStartScheduler.
Иерархически FreeRTOS содержит два уровня:

- Ядро системы: tasks.c, queue.c, list.c. Дополнительно для использования co-routine нужен файл croutine.c. Плюс несколько заголовочных файлов в папке include.
- Абстрактный уровень представления аппаратного обеспечения (Hardware Abstract Layer HAL) для комбинации компилятор-микроконтроллер. Этот уровень находится в фай-

лах port.c и portmacro.h. Этот уровень зависит от используемого микроконтроллера и компилятора.

Настройка ОС под конкретное приложение осуществляется в файле FreeRTOSConfig.h.

2 Компиляция

Компонетны, необходимые для компиляции FreeRTOS:

Linux

- MultiCletSDK
- GNU Make

Windows

- MultiCletSDK for Windows
- Утилита make из (MinGW или CygWin)

Настройка makefile.

- Откройте makefile текстовым редактором (например Notepad++, vim, nano ...)
- Укажите вашу ОС в переменной SYSTEM=(WIN32/LINUX)
- Укажите путь до компилятора в переменной LCCDIR

Компиляция демонстрационного проекта.

В загруженном с сайта архиве FreeRTOS реализован пример работы операционной системы с очередью на отладочной плате HW1-MCp04 (для запуска примера на LDM-MCp0411100101-Q208 Evolution, необходимо перенастроить UART в файле main.c). В данном примере создаётся очередь xQueue и 3 задачи: Sender1, Sender2, Receiver. Sender1 и Sender2, вставляют в очередь чило, переданное пользователем при создании задачи, а Receiver ожидает появление данных в очереди и выводит их.

Для того, чтобы собрать проект необходимо:

В ОС Linux:

- cd {rtos_dir}/Mcp4
- make all

В ОС Windows:

- Откройте текстовым редактором файл build_MinGW.cmd
- Укажите правильный путь до утилиты make.exe
- Запустите build_MinGW.cmd

Полученный образ (rtosdemo.bin) необходимо загрузить на плату. Результат работы программы можно наблюдать через UART.

3 Реализация

Стек для каждой задачи (т. е. функции) формируется в памяти, выделенной под эту задачу.

Вид стека:

Адрес BP задачи	
Адрес SP задачи	
pxCode	
pvParameters	
#GPR0 - #GPR7	
	<- Тут будет #SP
	<- Тут будет #BP
Стек задачи	

Файлы порта McP1, располагаются в {rtos_dir}/Source/portable/{compiler}/{platform}:

1. portmacro.h - файл прототипов и определений, конкретной архитектуры.
2. port.c - в этом файле содержится первичная инициализация стека для задачи, инициализация системного таймера, функции смены контекста в зависимости от типа планировщика.
3. critical.S - файл, в котором реализованы функции обеспечивающие безопасное нахождение в критических секциях
4. SRContext.S - файл с реализацией функций сохранения и восстановления контекста задачи.
5. vPortYield.S - принудительная (ручная) смена контекста.

Стартовый файл crt0.s, который проводит первичную инициализацию стека, создаёт таблицу векторов прерываний, первичный обработчик прерываний и т. д., располагается в {rtos_dir}/{platform}/lib

4 Конфигурирование

4.1 FreeRTOSConfig

Конфигурируемые параметры ядра FreeRTOS подстраиваются под каждое конкретное приложение. Эти параметры расположены в файле FreeRTOSConfig.h. Вот типичный пример, дальше будет описание каждого параметра.

```
#ifndef FREERTOS_CONFIG_H
#define FREERTOS_CONFIG_H
/* Это хорошее место для подключения заголовочных файлов, которые используются во
всём вашем приложении */
#include "something.h"

#define configUSE_PREEMPTION                    1
#define configUSE_IDLE_HOOK                     0
#define configUSE_TICK_HOOK                     0
#define configCPU_CLOCK_HZ                      80000000
#define configTICK_RATE_HZ                      10000
#define configMAX_PRIORITIES                    3
#define configMINIMAL_STACK_SIZE                128
#define configTOTAL_HEAP_SIZE                   2048
#define configMAX_TASK_NAME_LEN                 16
#define configIDLE_SHOULD_YIELD                 1
#define configUSE_MUTEXES                       0
#define configUSE_CO_ROUTINES                   0
#define configMAX_CO_ROUTINE_PRIORITIES         1

#define INCLUDE_vTaskPrioritySet                1
#define INCLUDE_uxTaskPriorityGet               1
#define INCLUDE_vTaskDelete                     1
#define INCLUDE_vTaskCleanUpResources           0
#define INCLUDE_vTaskSuspend                    1
#define INCLUDE_vResumeFromISR                  1
#define INCLUDE_vTaskDelayUntil                 1
#define INCLUDE_vTaskDelay                      1
#define INCLUDE_xTaskGetSchedulerState          1
#define INCLUDE_xTaskGetCurrentTaskHandle       1

#endif /* FREERTOS_CONFIG_H */
```

”config” параметры

configUSE_PREEMPTION	Установите 1 для вытесняющей многозадачности. 0 - для кооперативной многозадачности.
configUSE_IDLE_HOOK	Установите 1 для использования idle hook (описан выше), 0 - чтобы не использовать его.
configUSE_TICK_HOOK	Установите 1 для использования tick hook (описан выше), 0 - чтобы не использовать его.
configCPU_CLOCK_HZ	Частота внутреннего ядра процессора, который выполняет RTOS.
configTICK_RATE_HZ	Частота RTOS тика. Прерывание тика используется для измерения времени. Высокая частота увеличивает точность измерения времени. Но за увеличением частоты следует то, что ядро использует больше процессорного времени. Так же высокая частота приводит к снижению выделяемого процессорного времени каждой задаче.
configMAX_PRIORITIES	Количество приоритетов, доступных для задач. Любое количество задач может использовать один приоритет. Для сопрограмм см. дальше (configMAX_CO_ROUTINE_PRIORITIES). Каждый приоритет приводит к увеличению используемой ядром оперативной памяти.
configMINIMAL_STACK_SIZE	Минимальный размер стека задачи (portSTACK_TYPE).
configTOTAL_HEAP_SIZE	Общий объём оперативной памяти, доступный ядру (portCHAR).
configMAX_TASK_NAME_LEN	Максимальная допустимая длина имени задачи. Длина указывается включая терминирующий символ (\0).

configIDLE_SHOULD_YIELD	Этот параметр определяет поведение задач с приоритетом idle. Необходим если: 1. Используется вытесняющий планировщик. 2. Приложение создаёт задачи с приоритетом idle. Если configIDLE_SHOULD_YIELD установлен в 1 то задача idle сразу же уступит процессорное время, если какая-либо задача с idle приоритетом доступна для выполнения. Это гарантирует, что на idle тратится минимум времени, если есть другие доступные задачи. Установка configIDLE_SHOULD_YIELD в 0 предотвращает добровольную отдачу процессорного времени задачей idle. Это гарантирует, что всем задачам с приоритетом idle будет предоставлено одинаковое процессорное время. Но за счёт увеличения затрачиваемого времени на задачу idle.
configUSE_USE_MUTEXES	Установите 1, если используются мьютексы, иначе - 0.
configUSE_CO_ROUTINES	Установите 1, если используются сопрограммы, иначе - 0. Также, если используются сопрограммы, в сборку нужно включить файл croutine.c.
configMAX_CO_ROUTINE_PRIORITIES	Количество возможных приоритетов для сопрограмм. На одном приоритете может работать несколько сопрограмм.

Параметры INCLUDE

Макросы, начинающиеся с 'INCLUDE_' включают использование данного функционала. Если вы не используете данный функционал, то лучше отключить (установить равным 0) - это экономит память.
Каждый макрос такого вида:

```
INCLUDE_FunctionName
```

FunctionName - название API функции (или их набора), которые могут быть исключены из сборки. Чтобы включить эту функцию (или их набор), установите макрос равным 1, чтобы исключить - 0.

4.2 Управление памятью

Ядро должно выделять оперативную память каждой задаче, очереди и семафору (при создании). У различных встраиваемых систем реального времени могут быть различные способы организации оперативной памяти, поэтому единый алгоритм невозможен.

Для решения этой проблемы, в RTOS входит слой распределения памяти (функции vPortMalloc() и vPortFree()).

На данный момент существует четыре схемы распределения памяти, но для процессора Multiclet P1 используются только первые два. Каждая схема содержится в отдельном файле (heap_1.c, heap_2.c), которые расположены в {rtos_dir}/Source/portable/MemMang.

Первая схема: heap_1.c

Это самая простая схема из всех. Она не позволяет освобождать память, но, несмотря на это, подходит для большого числа приложений.
Алгоритм просто разбивает один массив на блоки меньшего размера при поступлении запросов на выделение памяти. Общим размером массива является configTOTAL_HEAP_SIZE, определённый в FreeRTOSConfig.h.
Такая схема:

- Может использоваться в приложениях, где никогда не вызываются функции vTaskDelete() и vQueueDelete().
- Всегда детермированна (вызов занимает всегда одно и тоже время)

heap_1.c подходит для большинства небольших систем реального времени, если они создают все задачи и очереди перед запуском ядра.

Вторая схема: heap_2.c

Эта схема использует улучшенный алгоритм, в отличии от первой схемы. Позволяет освобождать ранее выделенные блоки. Но не соединяет два смежных свободных блока в один большой.
Общий объём доступной оперативной памяти определяется
configTOTAL_HEAP_SIZE в FreeRTOSConfig.h.
Такая схема:

- Может использоваться, даже если приложение неоднократно вызывает vTaskCreate()/vTaskDelete() или vQueueCreate()/vQueueDelete() (делая множественные вызовы pvPortMalloc() и vPortFree())
- Не должна использоваться, если у удаляемых задач разная глубина стека, или удаляются очереди с разными длинами.
- Допускает фрагментацию, если ваше приложение должно создавать блоки очередей и задач в непредсказуемом порядке.
- Не детермированна

Эта схема подходит для большого числа небольших систем реального времени, которые должны динамически создавать задачи

heap_3.c и **heap_4.c** используют функции malloc() и free() из стандартной библиотеки.

5 Задачи и сопрограммы

5.1 Задачи

Приложение реального времени может быть структурировано как набор независимых задач. Каждая задача выполняется в своём собственном контексте. В рамках приложения одновременно может выполняться только одна задача, и планировщик отвечает за то, какие задачи когда должны выполняться. Поэтому планировщик может неоднократно приостанавливать и возобновлять задачу (переключать задачи и выходить) во время выполнения приложения. Так как задача не знает о деятельности планировщика, то он отвечает за сохранение контекста, чтобы задача была возобновлена в том же состоянии, что и была приостановлена. Для этого каждой задаче предоставляется отдельный стек. Когда задача приостановлена, контекст задачи хранится в стеке и может быть восстановлен перед возобновлением.

Task States (Состояния задачи)

- **Running** (запущена) - задача выполняется.
- **Ready** (готова) - задача может выполняться (т. е. не заблокирована и не ожидает), но не выполняется, т. к. выполняется задача с таким-же или выше приоритетом.
- **Blocked** (заблокирована) - задача ожидает временного или внешнего события. Например, если задача выполнила vTaskDelay(), она будет заблокирована до окончания задержки. Также задачи могут ожидать событий очереди или семафора. У блокировки всегда есть таймаут, после которого задача будет разблокирована. Заблокированные задачи недоступны для планирования.
- **Suspended** (приостановлена какой-либо задачей) - такие задачи также недоступны для планирования. Задачи будут входить/выходить из этого состояния при явном вызове vTaskSuspend() и xTaskResume() соответственно. Таймаут приостановленного состояния не задаётся.

Объявление задач

Задача должна иметь следующую структуру:

```
void vATaskFunction( void *pvParameters ){
 for( ;; )
 {
  //Код задачи
 }
}
```

Функции-задачи никогда не прерываются, поэтому обычно реализуются при помощи непрерывного цикла.

Задачи создаются с помощью xTaskCreate() и удаляются с помощью vTaskDelete()

Приоритеты задач

Каждой задаче назначается приоритет от 0 до (configMAX_PRIORITIES - 1).
configMAX_PRIORITIES объявляется в FreeRTOSConfig.h. Чем больше приоритетов, тем больше оперативной памяти потребляет ядро FreeRTOS.
Чем выше значение taskIDLE_PRIORITY (у создаваемой задачи), тем ниже приоритет. По умолчанию приоритет idle (taskIDLE_PRIORITY) равен нулю.
Планировщик гарантирует, что процессорное время отдаётся задаче с максимальным приоритетом из доступных (ready) задач. Даже, если задачи с высоким приоритетом тоже находятся в состоянии готовности (ready) длительное время.

Задача бездействия (простоя) (idle)

Задача idle автоматически создаётся при запуске планировщика. Она занимается освобождением памяти, выделенной задачам, которые были удалены. Поэтому в приложениях, использующих vTaskDelete(), важно обеспечить время для задачи idle. Утилита визуализации может быть использована для проверки времени, отводимого на idle.
У задачи idle нет других функций, поэтому ей может не предоставляться процессорное время при других условиях (если vTaskDelete() не использовалась)
Вполне возможно, что задачи приложения могут иметь приоритет ниже idle, тогда нужно повысить приоритет idle(taskIDLE_PRIORITY).

Перехватчик (hook) задачи idle

Перехватчик функции простоя (idle task hook) вызывается при каждом цикле задачи idle. Если вы хотите что-то делать во время простоя, есть два варианта:

1. Реализовать эту функциональность в idle task hook. Но она всегда должна быть доступна для выполнения, поэтому крайне важно не вызывать API функции, которые могут привести к блокировке (например, vTaskDelay(). Это допустимо для сопрограмм для блокировки в пределах функции-перехватчика)
2. Создание ещё одной задачи с приоритетом 0. Это более гибкое решение, но увеличивает расход оперативной памяти.

Для создания перехватчика:

1. Установите configUSE_IDLE_HOOK = 1 (в FreeRTOSConfig.h)
2. Объявите функцию с таким прототипом:

```
void vApplicationIdleHook( void );
```

Обычно это используют для того, чтобы вызывать режим энергосбережения процессора.

5.2 Сопрограммы

Сопрограммы концептуально схожи с задачами, но имеют следующие основные различия:

1. Все сопрограммы используют один стек на всех.
2. Сопрограммы используют приоритезированное кооперативное планирование по отношению к другим сопрограммам, а также могут быть включены в приложение, использующее вытесняющие задачи.
3. Сопрограмма реализуется через набор макросов.

6 Использование API

6.1 Общая информация

- В прерываниях должны использоваться только те API функции, которые предназначены для использования в прерываниях
- Задачи и сопрограммы используют разный API для доступа к очередям. Очереди не могут использоваться для связи между задачами и сопрограммами.
- Межзадачные коммуникации могут использоваться как с использованием полных API функций, так и функций предназначенных для прерываний (в названии имеют FromISR).

API в задаче

Задача может использовать любые API функции, кроме предназначенных для сопрограмм.

API в сопрограммах

В дополнение к специфическим для сопрограмм API, сопрограммы могут использовать следующие API вызовы:

- taskYIELD() - переключение задачи
- taskENTER_CRITICAL() - отключение обработки маскируемых прерываний
- taskEXIT_CRITICAL() - включение обработки маскируемых прерываний
- vTaskStartScheduler() - это по прежнему запускает планировщик, даже если задачи в приложении не используются
- vTaskSuspendAll() - блокирует планировщик
- xTaskResumeAll()
- xTaskGetTickCount()
- uxTaskGetNumberOfTasks()

6.2 Создание задач [task.h]

xTaskHandle - тип-ссылка на задачу (дескриптор задачи). Например, xTaskCreate возвращает (не через return, а через последний параметр) переменную типа xTaskHandle, которая может использоваться xTaskDelete для удаления задачи.

6.2.1 xTaskCreate

```
portBASE_TYPE xTaskCreate(
```

```
  pdTASK_CODE pvTaskCode,
  const portCHAR * const pcName,
  unsigned portSHORT usStackDepth,
  void *pvParameters,
  unsigned portBASE_TYPE uxPriority,
  xTaskHandle *pvCreatedTask
);
```

Создаёт новую задачу и помещает её в список доступных для выполнения задач.

pvTaskCode	Указатель на функцию-задачу. Функция-задача никогда не должна завершаться.
pcName	Описательное имя задачи. В основном используется при отладке. Максимальная длина (включая терминирующий символ) - configMAX_TASK_NAME_LEN
usStackDepth	Размер стека определяется как число переменных, которое он может хранить (а не байтов).
pvPatameters	Указатель, используемый как параметр для создаваемой задачи
uxPriority	Приоритет задачи
pvCreatedTask	Используется для возврата дескриптора созданной задачи

Возвращает:

pdPASS, если задача была успешно создана и помещена в список задач доступных для выполнения, иначе код ошибки, определённый в файле projdefs.h

Пример использования:

```
// Задача, которая будет создана
void vTaskCode(void *pvParameters){
 for (;;){
  // Код задачи
 }
}

// Функция, создающая задачу
void vOtherFunction(void){
 unsigned char ucParameterToPass;
 xTaskHandle xHandle;
 // Создаём задачу, её дескриптор сохраняем
 xTaskCreate(vTaskCode, "NAME", STACK_SIZE, &ucParameterToPass, tskIDLE_PRIORITY,
 &xHandle);
 // Используем дескриптор для удаления
 vTaskDelete(xHandle);
```

```
}
```

6.2.2 vTaskDelete

```
void vTaskDelete(xTaskHandle pxTask);
```

#define INCLUDE_vTaskDelete должен быть установлен в 1 для использования данной функции.

Задача будет удалена из всех списков: ready, blocked, suspended и event.

Примечание: За освобождение памяти ядра, выделенной удалённым задачам, отвечает задача idle. Поэтому очень важно, чтобы для задачи idle оставалось процессорное время (если вы используете vTaskDelete()). Память, выделенная задаче в процессе выполнения, автоматически не освобождается. Поэтому её нужно освободить в задаче перед её завершением.

pxTask	Дескриптор удаляемой задачи. Указание NULL ведёт к удалению задачи, из которой вызывается vTaskDelete(NULL)

Пример использования:

```
void vOtherFunction(void){
  xTaskHandle xHandle;
  // Создаём задачу
  xTaskCreate(vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle);
  // Используем дескриптор для удаления задачи
  vTaskDelete(xHandle);
}
```

6.3 Управление задачами [task.h]

6.3.1 vTaskDelay

```
void vTaskDelay(portTickType xTicksToDelay);
```

#define INCLUDE_vTaskDelay должен быть установлен в 1 для использования данной функции.

Задерживает задачу на определённое число тиков. Реальное время, на сколько будет приостановлена задача зависит от частоты тика. Константа portTICK_RATE_MS может использоваться для расчёта реального времени от количества тиков с точностью до периода одного тика.

xTicksToDelay	Количество тиков, на сколько блокируется задача.

Пример использования:

```
// Выполняет действие каждые 10 тиков

void vTaskFunction(void *pvParameters){
 portTickType xDelay, xNextTime;
 // Считаем время следующего запуска
 xNextTime = xTaskGetTickCount() + (portTickType) 10;
 for (;;){
  xDelay = xNextTime - xTaskGetTickCount();
  xNextTime +=(portTickType) 10;
  // Проверка на переполнение
  if (xDelay <=(portTickType) 10){
   vTaskDelay(xDelay);
  }
  // Код задачи
 }
}
```

6.3.2 vTaskDelayUntil

```
void vTaskDelayUntil(portTickType *pxPreviousWakeTime, portTickType
xTimeIncrement);
```

#define INCLUDE_vTaskDelayUntil должен быть установлен в 1 для использования данной функции.

Приостанавливает задачу до указанного времени. Эта функция может использоваться для циклических задач с постоянной частотой выполнения.
Эта функция отличается от vTaskDelay(). В vTaskDelay() указывается время, через которое нужно разблокировать задачу, а vTaskDelayUntil() указывается время, во сколько задачу нужно разблокировать.
vTaskDelay() блокирует задачу на определённое число тиков. Поэтому его трудно использовать для фиксированной частоты, т. к. задача может работать разное время из-за ветвлений. А в vTaskDelayUntil() указывается абсолютное время разблокировки.
Однако vTaskDelayUntil не будет блокировать задачу, если указано время пробуждения, которое уже прошло. Поэтому задача использующая vTaskDelayUntil() должна периодически пересчитывать необходимое время пробуждения, если она приостанавливается по каким-либо причинам, и при необходимости пропустить один или более циклов. Это может быть обнаружено сравнением переменной, передающейся по ссылке pxPreviousWakeTime с текущим значением счётчика тиков. Однако в большинстве случаев это не требуется.

Также можно использовать configTICK_RATE_MS.

pxPreviousWakeTime	Указатель на переменную, которая содержит время последней разблокировки. Переменная должна быть проинициализированна текущим временем. Переменная автоматически обновляется при каждом вызове vTaskDelayUntil()
xTimeIncrement	Период цикла. Задача будет разблокирована во время, равное (*pxPreviousWakeTime + xTimeIncrement). Вызов vTaskDelayUntil() с постоянным xTimeIncrement ведёт к постоянному периоду пробуждения задачи.

Пример:

```
// Выполняет действие каждые 10 тиков
void vTaskFunction(void *pvParameters){
  portTickType xLastWakeTime;
  const portTickType xFrequency = 10;
  // Инициализируем xLastWakeTime текущим временем
  xLastWakeTime = xTaskGetTickCount();
  for (;;){
    // Ждём следующего цикла
    vTaskDelayUntil(&xLastWakeTime, xFrequency);
    // Код задачи
  }
}
```

6.3.3 uxTaskPriorityGet

```
unsigned portBASE_TYPE uxTaskPriorityGet(xTaskHandle pxTask);
```

#define INCLUDE_vTaskPriorityGet должен быть установлен в 1 для использования данной функции.

Возвращает текущей приоритет задачи.

pxTask	Дескриптор задачи. Указание NULL ведёт к получению приоритета задачи, из которой вызывается vTaskPriorityGet(NULL)

Возвращает:

Приоритет задачи pxTask.

Пример:

```
void vAFunction(void){
  xTaskHandle xHandle;
```

```
  // Создаётся задача, дескриптор сохраняется.
  xTaskCreate(vTaskCode, "NAME", STACK_SIZE, NULL,  tskIDLE_PRIORITY, &xHandle);

  // Используем дескриптор для получения текущего приоритета задачи

  if (uxTaskPriorityGet(xHandle) != tskIDLE_PRIORITY){
    // Приоритет задачи был изменён
  }

  // Наш приоритет выше приоритета созданной задачи?
  if (uxTaskPriorityGet(xHandle) < uxTaskPriorityGet(NULL)){
    // Наш приоритет(полученный с помощью дескриптора NULL) выше.
  }
}
```

6.3.4 uxTaskPrioritySet

```
void vTaskPrioritySet(xTaskHandle pxTask, unsigned portBASE_TYPE uxNewPriority);
```

#define INCLUDE_vTaskPrioritySet должен быть установлен в 1 для использования данной функции.

Устанавливает приоритет задачи.
Контекст будет переключен до возврата, если устанавливаемый приоритет выше приоритета задачи, откуда меняется приоритет.

pxTask	Дескриптор задачи, приоритет которой будем менять. Если NULL - то задача, из которой вызывается vTaskPrioritySet()
uxNewPriority	Новый приоритет задачи

Пример:

```
void vAFunction(void){
 xTaskHandle xHandle;
 // Создаём задачу, дескриптор сохраняем
 xTaskCreate(vTaskCode, "NAME", STACK_SIZE, NULL,
 tskIDLE_PRIORITY, &xHandle);

 // Используем дескриптор для повышения приоритета.
 vTaskPrioritySet(xHandle, tskIDLE_PRIORITY + 1);

 // Используем NULL в качестве дескриптора для повышения приоритета текущей задачи
```

```
 vTaskPrioritySet(NULL, tskIDLE_PRIORITY + 1);
}
```

6.3.5 vTaskSuspend

```
void vTaskSuspend(xTaskHandle pxTaskToSuspend);
```

#define INCLUDE_vTaskSuspend должен быть установлен в 1 для использования данной функции.

Приостанавливает задачу.

pxTaskToSuspend	Дескриптор приостанавливаемой задачи. NULL означает задачу, из которой vTaskSuspend() была вызвана

Пример:

```
void vAFunction(void){
 xTaskHandle xHandle;
 // Создаём задачу
 xTaskCreate(vTaskCode, "NAME", STACK_SIZE, NULL,
 tskIDLE_PRIORITY, &xHandle);

 // Используем дескриптор для приостановки задачи
 vTaskSuspend(xHandle);

 // Созданная задача не будет работать пока
 // не вызвана функция vTaskResume(xHandle).

 // Приостанавливаем себя
 vTaskSuspend(NULL);
 // Пока кто-нибудь не разблокирует данную задачу через vTaskResume()

 // код размещённый в этом месте выполняться не будет.
}
```

6.3.6 vTaskResume

```
void vTaskResume(xTaskHandle pxTaskToResume);
```

#define INCLUDE_vTaskSuspend должен быть установлен в 1 для использования данной функции.

Возобновляет приостановленную задачу.
Задача приостановленная множеством вызовов vTaskSuspend() разблокируется одним вызовом vTaskResume()

pxTaskToResume	Дескриптор возобновляемой задачи.

Пример:

```
void vAFunction(void){
  xTaskHandle xHandle;
  // Создаём задачу.
  xTaskCreate(vTaskCode, "NAME", STACK_SIZE, NULL,
  tskIDLE_PRIORITY, &xHandle);

  // Используя дескриптор, приостанавливаем задачу
  vTaskSuspend(xHandle);

  // Созданная задача не будет работать
  // пока не вызвана функция vTaskResume(xHandle)...

  // Возобновляем приостановленную задачу
  vTaskResume(xHandle);
  // Созданная задача вновь будет работать
}
```

6.3.7 vTaskResumeFromISR

```
portBASE_TYPE vTaskResumeFromISR(xTaskHandle pxTaskToResume);
```

#define INCLUDE_vTaskSuspend и #define INCLUDE_xTaskResumeFromISR должны быть установлены в 1 для использования данных функций.

Функция для возобновления приостановленной задачи, причём эта функция может вызваться из прерывания (или сопрограммы). Задача, заблокированная вызовами vTaskSuspend(), будет доступна для работы после одного вызова vTaskResumeFromISR(). vTaskResumeFromISR() не надо использовать для синхронизации задач с прерыванием, если есть вероятность появления прерывания до приостановки задачи - это может привести к пропуску прерывания. Использование семафора для синхронизации позволит избежать таких проблем.

pxTaskToResume	Дескриптор возобновляемой задачи.

Возвращает:
pdTRUE, если возобновление задачи должно привести к переключению контекста,
иначе pdFALSE. Используется для определения необходимости переключения контекста.

Пример:

```
xTaskHandle xHandle;
void vAFunction(void){
  // Создаём задачу.
  xTaskCreate(vTaskCode, "NAME", STACK_SIZE, NULL,
  tskIDLE_PRIORITY, &xHandle);

}
void vTaskCode(void *pvParameters){
  // Задача приостанавливается и возобновляется
  for (;;){

    // Код функции

    // Задача самостоятельно приостанавливается.
    vTaskSuspend(NULL);

    // ждёт когда её возобновит прерывание
  }
}
void vAnExampleISR(void)
{
  portBASE_TYPE xYieldRequired;

  // Возобновляем приостановленную задачу
  xYieldRequired = xTaskResumeFromISR(xHandle);
  if (xYieldRequired == pdTRUE){

    // Мы должны переключить контекст
    portSwitchContext();
  }
}
```

6.4 Управление ядром [task.h]

6.4.1 Макросы

taskYIELD	Принудительное переключение контекста
taskENTER_CRITICAL	Макрос, отмечающий начало критической области кода.
taskEXIT_CRITICAL	Макрос, отмечающий завершение критической области кода.
taskDISABLE_INTERRUPTS	Отключение всех маскируемых прерываний.
taskENABLE_INTERRUPTS	Включение всех маскируемых прерываний.

6.4.2 vTaskStartScheduler

```
void vTaskStartScheduler(void);
```

Запускает обработку тиков RTOS. После вызова ядро контролирует какие и когда задачи выполняются.
Функция vTaskStartScheduler() может быть остановлена вызовом vTaskEndScheduler(). Также функция даёт сбой и мгновенно возвращается, если не хватает оперативной памяти для создания idle.

Пример:

```
void vAFunction(void){

  // Создаём задачу перед запуском ядра.
  xTaskCreate(vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, NULL);

  // Запускаем планировщик
  vTaskStartScheduler();

  // Сюда мы попадём, если планировщик не запустится.
}
```

6.4.3 vTaskEndScheduler

```
void vTaskEndScheduler(void);
```

Останавливает таймер RTOS. Все созданные задачи автоматически удаляются.
vTaskEndScheduler() выполняет все аппаратные действия, такие как остановка таймера.
vTaskEndScheduler() освободит все ресурсы выделенные ядру, но не освободит ресурсы, выделенные задачам (через явный вызов vPortMalloc())

Пример:

```
void vTaskCode(void * pvParameters){
  for (;;){
    // Код задачи
    vTaskEndScheduler();
  }
}

void vAFunction(void){
  // Создаём задачу перед запуском ядра
  xTaskCreate(vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, NULL);
  // Запускаем планировщик.
  vTaskStartScheduler();
  // Мы окажемся здесь только когда vTaskCode() вызовет
  // vTaskEndScheduler().
}
```

6.4.4 vTaskSuspendAll

```
void vTaskSuspendAll(void);
```

Приостанавливает всю активность ядра RTOS.
От вызова vTaskSuspendAll() до вызова xTaskResumeAll() задача будет выполняться без риска быть вытесненной.

Пример:

```
void vTask1(void * pvParameters){
  for (;;){
    // Код задачи

    // Предотвращаем вытеснение задачи ядром
    vTaskSuspendAll();
    // Выполняем операцию. Здесь нет необходимости использовать критические
    // секции.
    // В это время прерывания продолжают работать и счётчик тиков
    // тоже работает

    // Выполнение завершено, перезапускаем ядро
    xTaskResumeAll();
  }
```

```
}
```

6.4.5 xTaskResumeAll

```
portBASE_TYPE xTaskResumeAll(void);
```

Возобновляет работу ядра RTOS приостановленную vTaskSuspendAll(). После вызова xTaskResumeAll() ядро опять будет контролировать задачи.

Возвращает:
pdTRUE, если возобновление работы планировщика вызвало переключение контекста, иначе pdFALSE.

Пример:

```
void vTask1(void * pvParameters){
  for (;;){
    // Код задачи.

    // Предотвращаем вытеснение задачи ядром.
        vTaskSuspendAll();
    // Выполняем операцию.
    // В это время прерывания продолжают работать и счётчик тиков
    // тоже работает

    // Операция завершена. Перезапускаем ядро
    // Принудительно переключаем контекст, если это не сделал планировщик.
    if (!xTaskResumeAll()){
      taskYIELD();
    }
  }
}
```

6.4.6 xTaskGetCurrentTaskHandle

```
xTaskHandle xTaskGetCurrentTaskHandle(void);
```

#define INCLUDE_xTaskGetCurrentTaskHandle должен быть установлен в 1 для использования этой функции.

Возвращает:
Дескриптор задачи, из которой вызывается xTaskGetCurrentTaskHandle().

6.4.7 xTaskGetTickCount

```
volatile portTickType xTaskGetTickCount(void);
```

#define INCLUDE_xTaskGetSchedulerState должен быть установлен в 1 для использования этой функции.

Возвращает:
Количество тиков с момента запуска планировщика.

6.4.8 xTaskGetSchedulerState

```
portBASE_TYPE xTaskGetSchedulerState(void);
```

Возвращает:
Одну из следующих констант [из task.h]: taskSCHEDULER_NOT_STARTED, taskSCHEDULER_RUNNING, taskSCHEDULER_SUSPENDED.

6.4.9 uxTaskGetNumberOfTasks

```
unsigned portBASE_TYPE uxTaskGetNumberOfTasks(void);
```

Возвращает:
Количество задач, которыми управляет ядро RTOS. Включает следующие задачи: доступные для выполнения, заблокированные и приостановленные. Удалённые задачи, но ресурсы которых функция idle не освободила, также учитывается.

6.4.10 vTaskList

```
void vTaskList(portCHAR *pcWriteBuffer);
```

#define configUSE_TRACE_FACILITY, #define INCLUDE_vTaskDelete и
#define INCLUDE_vTaskSuspend
должны быть установлены в 1 для использования этих функций.
Примечание: эта функция отключает прерывания на время работы. Она не должна использоваться в рабочем приложении. Она используется только для отладки.

Перечисляет все задачи, их текущее состояние и максимальный уровень использования стека. Состояние задачи обозначается так:

- Blocked (заблокирована)
- Ready (готова для выполнения)
- Deleted (удалена)

- Suspended (приостановлена)

pcWriteBuffer	Буфер, в который пишется информация в кодировке ASCII. Этот буфер должен быть достаточно большим, чтобы поместить весть отчёт. Требуется около 40 байт на задачу

6.4.11 vTaskStartTrace

```
void vTaskStartTrace(portCHAR * pcBuffer, unsigned portLONG ulBufferSize);
```

Запускает трассировку.

pcBuffer	Буфер, в который пишется трассировка
ulBufferSize	Размер буфера в байтах. Трассировка идёт до полного заполнения буфера либо вызова ulTaskEndTrace()

6.4.12 ulTaskEndTrace

```
unsigned portLONG ulTaskEndTrace(void);
```

Останавливает трассировку.

Возвращает:
Количество байт, записанных в буфер.

6.5 Управление очередями [queue.h]

6.5.1 uxQueueMessagesWaiting

```
unsigned portBASE_TYPE uxQueueMessagesWaiting(xQueueHandle xQueue);
```

Возвращает количество элементов в очереди.

xQueue	Дескриптор очереди

6.5.2 vQueueDelete

```
void vQueueDelete(xQueueHandle xQueue);
```

Удаляет очередь - освобождает всю память, выделенную под хранение элементов в очереди.

xQueue	Дескриптор удаляемой очереди

6.5.3 xQueueCreate

```
xQueueHandle xQueueCreate(unsigned portBASE_TYPE uxQueueLength,
unsigned portBASE_TYPE uxItemSize);
```

Создаёт новую очередь и возвращает её дескриптор.

uxQueueLength	Максимальное количество элементов, сколько может вместить очередь
uxItemSize	Количество байт под каждый элемент.

Возвращает:

Если очередь создана, то её дескриптор, иначе 0.

Пример:

```
struct AMessage {
        portCHAR ucMessageID;
        portCHAR ucData[ 20 ];
};
void vATask(void *pvParameters){
        xQueueHandle xQueue1, xQueue2;
        // Создаёт очередь, вмещающую 10 элементов типа unsigned long.
        xQueue1 = xQueueCreate(10, sizeof(unsigned portLONG));
        if (xQueue1 == 0){
                // Очередь не создана и не может использоваться.
        }
        // Создаёт очередь, вмещающую 10 указателей на структуры AMessage.
        xQueue2 = xQueueCreate(10, sizeof(struct AMessage *));
        if (xQueue2 == 0){
                // Очередь не создана и не может использоваться.
        }
        // Остальной код
}
```

6.5.4 xQueueSend

```
portBASE_TYPE xQueueSend(
                            xQueueHandle xQueue,
                              const void *pvItemToQueue,
                            portTickType xTicksToWait
)
```

Этот макрос вызывает xQueueGenericSend(). xQueueSend() эквивалентен xQueueSendToBack(). Отправляет копию объекта в очередь. Эту функцию нельзя использовать в прерываниях. Для этого есть xQueueSendFromISR().

xQueue	Дескриптор очереди, в которую будет произведена запись
pvItemToQueue	Указатель на данные, помещаемые в очередь. Количество байт было указано при создании очереди.
xTicksToWait	Максимальное время (в тиках) ожидания места в очереди. Если 0, то при отсутствии места сразу возвращает errQUEUE_FULL. Если INCLUDE_vTaskSuspend=1, то можно указать portMAX_DELAY чтобы можно было ждать бесконечно много времени

Возвращает:
pdTRUE, если успешно отправлено, иначе errQUEUE_FULL.

Пример:

```
struct AMessage{
        portCHAR ucMessageID;
        portCHAR ucData[ 20 ];
} xMessage;

unsigned portLONG ulVar = 10UL;

void vATask(void *pvParameters){
        xQueueHandle xQueue1, xQueue2;
        struct AMessage *pxMessage;
        // Создаём очередь, вмещающую 10 элементов типа unsigned long.
        xQueue1 = xQueueCreate(10, sizeof(unsigned portLONG));
        // Создаём очередь, вмещающую 10 указателей на AMessage.
        xQueue2 = xQueueCreate(10, sizeof(struct AMessage *));

        if (xQueue1 != 0){
                // Отправляем unsigned long. Ожидаем места максимум 10 тиков
                if (xQueueSend(xQueue1,(void *) &ulVar,
               (portTickType) 10) != pdPASS){
                        // Не получилось.
                }
        }
        if (xQueue2 != 0){
                // Отправляем указатель на объект типа AMessage.
                // Если места нет, то не ждём.
```

```
                pxMessage = & xMessage;
                xQueueSend(xQueue2,(void *) &pxMessage,(portTickType) 0);
        }
        // Остальной код
}
```

6.5.5 xQueueSendToBack

```
portBASE_TYPE xQueueSendToBack(xQueueHandle xQueue,
                                    const void *pvItemToQueue,
                                   portTickType xTicksToWait
);
```

Этот макрос вызывает xQueueGenericSend(). Это эквивалентно вызову xQueueSend().

6.5.6 xQueueSendToFront

```
portBASE_TYPE xQueueSendToToFront(xQueueHandle xQueue,
                                      const void *pvItemToQueue,
                                     portTickType xTicksToWait
);
```

Этот макрос вызывает xQueueGenericSend(). Отправляет копию объекта в начало очереди. Эту функцию нельзя использовать в прерываниях. Для этого есть xQueueSendToFrontFromISR().

xQueue	Дескриптор очереди, в которую будет произведена запись.
pvItemToQueue	Указатель на данные, помещаемые в очередь. Количество байт было указано при создании очереди.
xTicksToWait	Максимальное время (в тиках) ожидания места в очереди. Если 0, то при отсутствии места сразу возвращает errQUEUE_FULL. Если INCLUDE_vTaskSuspend=1, то можно указать portMAX_DELAY чтобы можно было ждать бесконечно много времени

Возвращает:
pdTRUE, если успешно отправлено, иначе errQUEUE_FULL.

Пример:

```
struct AMessage{
portCHAR ucMessageID;
portCHAR ucData[ 20 ];
} xMessage;
unsigned portLONG ulVar = 10UL;
```

```
void vATask(void *pvParameters){
        xQueueHandle xQueue1, xQueue2;
        struct AMessage *pxMessage;
        // Создаём очередь, вмещающую 10 элементов типа unsigned long.
        xQueue1 = xQueueCreate(10, sizeof(unsigned portLONG));

        // Создаём очередь, вмещающую 10 указателей на AMessage.
        xQueue2 = xQueueCreate(10, sizeof(struct AMessage *));
        // ...
        if (xQueue1 != 0)
        {
                // Отправляем unsigned long. Ожидаем место максимум 10 тиков
                if (xQueueSendToFront(xQueue1,
              (void *) &ulVar,(portTickType) 10) != pdPASS){
                        // Не получилось.
                }
        }
        if (xQueue2 != 0)
        {
                // Отправляем указатель на объект типа AMessage.
                // Если места нет, то не ждём.
                pxMessage = & xMessage;
                xQueueSendToFront(xQueue2,(void *) &pxMessage,
              (portTickType) 0);
        }
        // Остальной код
}
```

6.5.7 xQueueReceive

```
portBASE_TYPE xQueueReceive(xQueueHandle xQueue,
                                    void *pvBuffer,
                            portTickType xTicksToWait
);
```

Этот макрос вызывает xQueueGenericReceive().
Получает копию элемента из очереди в буфер. Из очереди элемент удаляется.

Эту функцию нельзя использовать в прерываниях. Для этого есть xQueueReceiveFromISR.

xQueue	Дескриптор очереди, из которой будет произведено чтение
pvBuffer	Указатель на буфер, в который будет записан прочитанный элемент
xTicksToWait	Максимальное время(в тиках) ожидания элемента в очереди. Если INCLUDE_vTaskSuspend=1, то можно указать portMAX_DELAY чтобы можно было ждать бесконечно много времени

Возвращает:
pdTRUE, если успешно прочитано, иначе pdFALSE

Пример:

```
struct AMessage {
        portCHAR ucMessageID;
        portCHAR ucData[ 20 ];
} xMessage;
xQueueHandle xQueue;

// Задача, создающая очередь и отправляющая туда сообщение
void vATask(void *pvParameters){
        struct AMessage *pxMessage;
        // Создаём очередь, вмещающую 10 указателей на AMessage.
        xQueue = xQueueCreate(10, sizeof(struct AMessage *));
        if (xQueue == 0){
                // Не получилось создать очередь
        }

        // Отправляем указатель на объект AMessage.
        pxMessage = & xMessage;
        xQueueSend(xQueue,(void *) &pxMessage,(portTickType) 0);
        // Остальной код
}

// Задача получающая сообщения из очереди
void vADifferentTask(void *pvParameters){
        struct AMessage *pxRxedMessage;
        if (xQueue != 0){
                // Получаем сообщение из очереди. Если сообщения нет, то ожидаем
                // его 10 тиков
                if (xQueueReceive(xQueue, &(pxRxedMessage),
              (portTickType) 10)){
                        // pcRxedMessage теперь указывает на объект AMessage,
```

```
                        // отправленный vATask
                }
        }
        // Остальной код задачи
}
```

6.5.8 xQueuePeek

```
portBASE_TYPE xQueuePeek(xQueueHandle xQueue,
                                    void *pvBuffer,
                            portTickType xTicksToWait
);
```

Этот макрос вызывает xQueueGenericReceive().
Получает элемент из очереди без его последующего удаления. Полученный элемент копируется в буфер.
Полученные элементы опять будут прочитаны при следующем вызове xQueuePeek и xQueueReceive().
Этот макрос нельзя использовать в прерываниях.

xQueue	Дескриптор очереди, откуда будет производиться чтение
pvBuffer	Указатель на буфер для чтения. Он должен вмещать размер элемента очереди
xTicksToWait	Максимальное время(в тиках) ожидания элемента в очереди. Если INCLUDE_vTaskSuspend = 1, то можно указать portMAX_DELAY чтобы ждать бесконечно много времени.

Возвращает:
pdTRUE, если успешно прочитано, иначе errQUEUE_FULL.

Пример:

```
struct AMessage{
        portCHAR ucMessageID;
        portCHAR ucData[ 20 ];
} xMessage;
xQueueHandle xQueue;

// Задача, создающая очередь, и отправляющая туда сообщения
void vATask(void *pvParameters){
        struct AMessage *pxMessage;
        // Создаём очередь, вмещающую 10 указателей на AMessage.
        xQueue = xQueueCreate(10, sizeof(struct AMessage *));
```

```
        if (xQueue == 0){
                // Не получилось создать очередь
        }
        // Отправляем указатель на объект AMessage.
        pxMessage = & xMessage;
        xQueueSend(xQueue,(void *) &pxMessage,(portTickType) 0);
        // Остальной код
}

// Задача, просматривающая очередь
void vADifferentTask(void *pvParameters){
        struct AMessage *pxRxedMessage;
        if (xQueue != 0){
           // Просматриваем сообщение в очереди. Если сообщение сейчас недоступно,
           // ожидаем в течение 10 тиков
           if (xQueuePeek(xQueue, &(pxRxedMessage),(portTickType) 10)){
                // pcRxedMessage теперь указывает на структуру AMessage
                // Но элемент в очереди остался
           }
        }
        // Остальной код
}
```

6.5.9 xQueueSendFromISR

```
portBASE_TYPE xQueueSendFromISR(xQueueHandle pxQueue,
                                const void *pvItemToQueue,
                               portBASE_TYPE xTaskPreviouslyWoken
);
```

Этот макрос вызывает xQueueGenericSendFromISR(). Он оставлен для обратной совместимости с предыдущими версиями, у которых нет макросов xQueueSendToBackFromISR() и xQueueSendToFrontFromISR().
Отправляет элемент в очередь. Этот макрос можно использовать в прерываниях.
Элементы копируются (а не передаются по ссылке).

xQueue	Дескриптор очереди, в которую будет произведена запись
pvItemToQueue	Указатель на данные, помещаемые в очередь.
xTaskPreviouslyWoken	Используется чтобы разбудить только одну задачу из прерывания (например, если несколько задач читают из одной очереди). Первому вызову должно передаваться pdFALSE, а последующим - результат предыдущего вызова).

Возвращает:
pdTRUE, если отправка разбудила какую-либо задачу. Используется для определения необходимости переключения контекста

6.5.10 xQueueSendToBackFromISR

```
portBASE_TYPE xQueueSendToBackFromISR(xQueueHandle pxQueue,
                                      const void *pvItemToQueue,
                                    portBASE_TYPE xTaskPreviouslyWoken
);
```

Этот макрос вызывает xQueueGenericSendFromISR().
Отправляет элемент в конец очереди. Этот макрос можно использовать в прерываниях.
Элементы копируются (а не передаются по ссылке).

xQueue	Дескриптор очереди, в которую будет произведена запись
pvItemToQueue	Указатель на данные, помещаемые в очередь.
xTaskPreviouslyWoken	Используется чтобы разбудить только одну задачу из прерывания (например, если несколько задач читают из одной очереди). Первому вызову должно передаваться pdFALSE, а последующим - результат предыдущего вызова.

Возвращает:
pdTRUE, если отправка разбудила какую-либо задачу. Используется для определения необходимости переключения контекста.

6.5.11 xQueueSendToFrontFromISR

```
portBASE_TYPE xQueueSendToFrontFromISR(xQueueHandle pxQueue,
                                       const void *pvItemToQueue,
                                     portBASE_TYPE xTaskPreviouslyWoken
);
```

Этот макрос вызывает xQueueGenericSendFromISR().
Отправляет элемент в начало очереди. Этот макрос можно использовать в прерываниях.
Элементы копируются (а не передаются по ссылке).

xQueue	Дескриптор очереди, в которую будет произведена запись
pvItemToQueue	Указатель на данные, помещаемые в очередь.
xTaskPreviouslyWoken	Используется чтобы разбудить только одну задачу из прерывания (например, если несколько задач читают из одной очереди). Первому вызову должно передаваться pdFALSE, а последующим - результат предыдущего вызова.

Возвращает:
pdTRUE, если отправка разбудила какую-либо задачу. Используется для определения необходимости переключения контекста.

6.5.12 xQueueReceiveFromISR

```
portBASE_TYPE xQueueReceiveFromISR(xQueueHandle pxQueue,
                                                    void *pvBuffer,
                                   portBASE_TYPE *pxTaskWoken
);
```

Получает элемент из очереди. Эту функцию можно использовать в прерываниях.

xQueue	Дескриптор очереди, из которой будем читать
pvBuffer	Указатель на буфер для чтения
pxTaskWoken	Задача может быть заблокирована во время ожидания места в очереди. Если этот вызов был причиной разблокировки, то *pxTaskWoken будет установлен в pdTRUE, иначе он останется без изменений

Возвращает:
pdTRUE, если успешно прочитано, иначе pdFALSE.

Пример:

```
xQueueHandle xQueue;
// Функция, создающая очередь, и отправляющая туда числа
void vAFunction(void *pvParameters){
        portCHAR cValueToPost;
        const portTickType xBlockTime =(portTickType)0xff;
        // Создаём очередь, вмещающую 10 символов
        xQueue = xQueueCreate(10, sizeof(portCHAR));
        if (xQueue == 0){
                // Не получилось
        }

        // Отправляем символы в прерывание. Если очередь заполнена,
```

```
        // то ждём в течение xBlockTime тиков.
        cValueToPost = 'a';
        xQueueSend(xQueue,(void *) &cValueToPost, xBlockTime);
        cValueToPost = 'b';
        xQueueSend(xQueue,(void *) &cValueToPost, xBlockTime);
        // ... продолжаем отправлять символы...
        // задача заблокируется при заполнении очереди
        cValueToPost = 'c';
        xQueueSend(xQueue,(void *) &cValueToPost, xBlockTime);
}

// прерывание, выводящее символы из очереди
void vISR_Routine(void){
        portBASE_TYPE xTaskWokenByReceive = pdFALSE;
        portCHAR cRxedChar;
        while (xQueueReceiveFromISR(xQueue,(void *) &cRxedChar,
        &xTaskWokenByReceive)){
            // Получили символ, выводим его.
            vOutputCharacter(cRxedChar);
            /* Если была разбужена задача (из-за освобождения места в очереди)
            xTaskWokenByReceive
            будет установлен в pdTRUE. Независимо от того, сколько символов было
            прочитано, разбужена будет только одна задача */
        }
        if (xTaskWokenByPost != pdFALSE){
            // Если надо, переключаем контекст
            taskYIELD();
        }
}
```

6.6 Семафоры [semphr.h]

6.6.1 SemaphoreCreateBinary

`vSemaphoreCreateBinary(xSemaphoreHandle xSemaphore)`

Этот макрос создаёт семафор, используя существующую реализацию очереди. Длина очереди равна 1, т. к. это двоичный семафор. Размер данных равен 0, т. к. мы не хотим хранить там данные - мы просто хотим знать пуста очередь или нет.
Бинарные семафоры и мьютексы очень схожи, но имеют некоторые тонкие отличия: мьютек-

сы включают механизм наследования приоритета, а бинарные семафоры - нет. Это делает бинарные семафоры удобными для синхронизации (между задачами или между задачами и прерываниями), а мьютексы - для простого взаимного исключения.

Мьютексы и двоичные семафоры относятся к переменным типа xSemaphoreHandle и могут быть использованы любой функцией, принимающей данный тип.

xSemaphore	Дескриптор созданного семафора (записывается макросом в переменную), должен иметь тип xSemaphoreHandle

Пример:

```
xSemaphoreHandle xSemaphore;
void vATask(void * pvParameters){

  // Создаём семафор
  vSemaphoreCreateBinary(xSemaphore);
  if (xSemaphore != NULL)
  {
    // Семафор успешно создан
    // Теперь его можно использоваеть
  }
}
```

6.6.2 xSemaphoreCreateMutex

```
xSemaphoreHandle xQueueCreateMutex(void)
```

Этот макрос создаёт мьютекс, используя существующую реализацию очереди.

См. SemaphoreCreateBinary()

Возвращает:
Дескриптор созданного мьютекса.

Пример:

```
xSemaphoreHandle xMutex;
void vATask(void * pvParameters){

  // Создаём мьютекс
  xMutex = vSemaphoreCreateMutex();
  if (xMutex != NULL){
    // Мьютекс успешно создан
    // И может использоваться
```

```
  }
}
```

6.6.3 xSemaphoreTake

`xSemaphoreTake(xSemaphoreHandle xSemaphore, portTickType xBlockTime)`

Макрос для получения семафора. Семафор должен быть предварительно создан.
Этот макрос нельзя вызывать из обработчика прерывания. Для этих целей можно воспользоваться xQueueReceiveFromISR.

xSemaphore	Дескриптор уже существующего семафора
xBlockTime	Время(в тактах), сколько ждать доступности семафора. portTICK_RATE_MS можно использовать.

Если INCLUDE_vTaskSuspend установлен в 1 и xBlockTime указано в port_MAX_DELAY, то задача блокируется на неопределённое время, пока не будет доступен семафор.

Возвращает:
pdTRUE, если семафор был получен; pdFALSE, если время xBlockTime истекло, а семафор так и не стал доступным.

Пример:

```
xSemaphoreHandle xSemaphore = NULL;

// Задача, создающая семафор
void vATask(void * pvParameters){

  // Создаётся мьютекс для защиты общих ресурсов.
  xSemaphore = xSemaphoreCreateMutex();
}

// Задача, также использующая семафор(мьютекс)
void vAnotherTask(void * pvParameters){

  // Код задачи.

  if (xSemaphore != NULL){
    // Пытаемся получить семафор. Таймаут - 10 тиков
    if (xSemaphoreTake(xSemaphore,(portTickType) 10) == pdTRUE){
      // Мы получили семафор, поэтому спокойно работаем
      // с общим ресурсом
```

```
      // ...
      // После окончания работы с общим ресурсом
      // освобождаем семафор
      xSemaphoreGive(xSemaphore);
    }
    else
    {
      // Мы не смогли получить семафор, поэтому не можем безопасно работать с
      // общими ресурсами
    }
  }
}
```

6.6.4 xSemaphoreGive

`xSemaphoreGive(xSemaphoreHandle xSemaphore)`

Этот макрос возвращает семафор.
Этот макрос нельзя использовать из прерывания. Для этого есть xSemaphoreGiveFromISR().

xSemaphore	Дескриптор уже существующего семафора.

Возвращает:
pdTRUE - в случае успеха; pdFALSE, если произошла ошибка.

Пример:

```
xSemaphoreHandle xSemaphore = NULL;
void vATask(void * pvParameters){

        // Создаём мьютекс для защиты общих ресурсов.
        xSemaphore = xSemaphoreCreateMutex();
        if (xSemaphore != NULL){
                if (xSemaphoreGive(xSemaphore) != pdTRUE){
                        // Мы окажемся тут, т.к. никто семафор ещё не трогал
                }
                // Получаем семафор, если он доступен прямо сейчас
                if (xSemaphoreTake(xSemaphore,(portTickType) 0)){
                        // Теперь у нас есть семафор, и мы может
                        // работать с общими ресурсами
                        // ...
                        // После окончания работы надо положить семафор на место
```

```
                        if (xSemaphoreGive(xSemaphore) != pdTRUE)
                        {

                        }
                }
        }
}
```

6.6.5 xSemaphoreGiveFromISR

`xSemaphoreGiveFromISR(xSemaphoreHandle xSemaphore, portBASE_TYPE xTaskPreviouslyWoken)`

Возвращает существующий семафор.
Этот макрос может использоваться из обработчика прерывания.

xSemaphore	Дескриптор уже существующего семафора.
xTaskPreviouslyWoken	Используется для нескольких вызовов xSemaphoreGiveFromISR() из одного прерывания. Первому вызову всегда должно передаваться pdFALSE. Следующими вызовы должны включать результат выполнения предыдущего

Возвращает:
pdTRUE, если возвратом семафора стала доступной для выполнения какая-либо задача. Используется для определения необходимости принудительного переключения контекста.

Пример:

```
#define LONG_TIME 0xffff
#define TICKS_TO_WAIT 10
xSemaphoreHandle xSemaphore = NULL;

// Повторяющаяся задача
void vATask(void * pvParameters){
    // Мы используем семафоры для синхронизации. Поэтому используем бинарный семафор
    // а не мьютекс. Мы должны убедиться, что семафор не будет использоваться
    // прерыванием, пока мы не создадим этот семафор.
    xSemaphoreCreateBinary(xSemaphore);
    for (;;){
            // Мы хотим выполнять эту задачу каждые 10 тиков таймера.
            // Ждём семафор
            if (xSemaphoreTake(xSemaphore, LONG_TIME) == pdTRUE){

    // Выполнение задачи.
```

```
    // ...
                    // Мы закончили нашу задачу. Цикл нас вернёт обратно, где мы
                    // будем ждать очередного семафора.
            }
    }
}

// Обработчик прерывания таймера
void vTimerISR(void * pvParameters){
        static unsigned portCHAR ucLocalTickCount = 0;
        static portBASE_TYPE xTaskWoken = pdFALSE;
        // Произошло срабатывание таймера

        // Пора запустить vATask?
        ucLocalTickCount++;
        if (ucLocalTickCount >= TICKS_TO_WAIT){
                // Разблокируем задачу путём отдавания семафора
                xTaskWoken = xSemaphoreGiveFromISR(xSemaphore, xTaskWoken);

                // Сбрасываем счётчик, таким образом мы установим новый семафор
                // через 10 тиков.
                ucLocalTickCount = 0;
        }
        // Если xTaskWoken, то вы можете принудительно переключить контекст здесь
}
```

6.7 Специфический API для сопрограмм [croutine.h]

xCoRoutineHandle - тип, используемый для ссылки на сопрограмму. Автоматически передаётся в каждую сопрограмму.

6.7.1 xCoRoutineCreate

```
portBASE_TYPE xCoRoutineCreate(crCOROUTINE_CODE pxCoRoutineCode,
                               unsigned portBASE_TYPE uxPriority,
                               unsigned portBASE_TYPE uxIndex);
```

Создаёт сопрограмму и добавляет её в список доступных для выполнения.

pxCoRoutineCode	Указатель на функцию-сопрограмму
uxPriority	Приоритет сопрограммы. Этот приоритет имеет смысл только между сопрограммами
uxIndex	Используется для различия между разными сопрограммами, основанными на одной функции.

Возвращает:
pdPASS, если сопрограмма была успешно создана и добавлена в список, иначе ошибку из ProjDefs.h.

6.7.2 crDELAY

```
void crDELAY(xCoRoutineHandle xHandle, portTickType xTicksToDelay)
```

crDELAY - макрос, который задерживает сопрограмму на определённое время.
crDELAY может вызываться только из задерживаемой сопрограммы и ниоткуда больше, т. к. у сопрограммы нет собственного стека.

pxCoRoutineCode	Дескриптор задерживаемой сопрограммы.
xTickToDelay	Количество тиков для ожидания. Аналогично единственному аргументу vTaskDelay() для задач.

6.7.3 crQUEUE_SEND

```
crQUEUE_SEND(xCoRoutineHandle xHandle,
                  xQueueHandle pxQueue,
                          void *pvItemToQueue,
                  portTickType xTicksToWait,
                 portBASE_TYPE *pxResult
)
```

crQUEUE_SEND - отправка элемента в очередь.
crQUEUE_SEND() и crQUEUE_RECEIVE являются аналогами функций xQueueSend(), xQueueReceive(), но для сопрограмм. Очереди нельзя использовать для коммуникаций между задачей и сопрограммой (и наоборот).
crQUEUE_SEND() может вызываться только из самой сопрограммы.

xHandle	Дескриптор вызывающей сопрограммы.
pxQueue	Дескриптор существующей очереди, в которую будет произведена запись.
pvItemToQueue	Указатель на данные, помещаемые в очередь.
xTickToDelay	Количество тиков, сколько можно ждать свободного места в очереди.
pxResult	Указатель на переменную. В неё записывается pdPASS, если всё прошло успешно, иначе - ошибка.

6.7.4 crQUEUE_RECEIVE

```
void crQUEUE_RECEIVE(xCoRoutineHandle xHandle,
                             xQueueHandle pxQueue,
                                     void *pvBuffer,
                             portTickType xTicksToWait,
                            portBASE_TYPE *pxResult
)
```

crQUEUE_RECEIVE - достать элемент из очереди.
crQUEUE_SEND() и crQUEUE_RECEIVE являются аналогами функций xQueueSend(), xQueueReceive(). Очереди нельзя использовать для коммуникаций между задачей и сопрограммой (и наоборот).
crQUEUE_RECEIVE() может вызываться только из самОй сопрограммы, и не из вызываемых оттуда функций. Это так потому, что сопрограмма не имеет своего стека.

xHandle	Дескриптор вызывающей сопрограммы.
pxQueue	Дескриптор существующей очереди, из которой будет произведено чтение
pvBuffer	Буфер, куда будет производиться чтение.
xTickToDelay	Количество тиков, сколько можно ждать появление объекта в очереди
pxResult	Указатель на переменную. В неё записывается pdPASS, если всё прошло успешно, иначе - ошибка.

6.7.5 crQUEUE_SEND_FROM_ISR

```
portBASE_TYPE crQUEUE_SEND_FROM_ISR(xQueueHandle pxQueue,
                                                void *pvItemToQueue,
                                    portBASE_TYPE xCoRoutinePreviouslyWoken
)
```

crQUEUE_SEND_FROM_ISR() - помещение элемента в очередь. Используется из прерывания.

Макросы crQUEUE_SEND_FROM_ISR() и crQUEUE_RECEIVE_FROM_ISR() эквивалентны xQueueSendFromISR() и xQueueReceiveFromISR(), но используются сопрограммами.

pxQueue	Дескриптор существующей очереди, в которую будет произведена запись
pvItemToQueue	Указатель на данные, помещаемые в очередь.
xCoRoutinePreviouslyWoken	Используется для нескольких вызовов crQUEUE_SEND_FROM_ISR() из одного прерывания. Первому вызову всегда должно передаваться pdFALSE. Следующим вызовам должен передаваться результат выполнения предыдущего вызова.

Возвращает:

pdTRUE, если записью в очередь стала доступной для выполнения какая-либо задача. Используется для определения необходимости принудительного переключения контекста.

6.7.6 crQUEUE_RECEIVE_FROM_ISR

```
portBASE_TYPE crQUEUE_SEND_FROM_ISR(xQueueHandle pxQueue,
                                    void *pvBuffer,
                                portBASE_TYPE *pxCoRoutineWoken
                                )
```

Макросы crQUEUE_SEND_FROM_ISR() и crQUEUE_RECEIVE_FROM_ISR() эквивалентны xQueueSendFromISR() и xQueueReceiveFromISR(), но используются сопрограммами.

pvBuffer	Указатель на буфер, в который будет произведено чтение.
xCoRoutineWoken	Какая-либо сопрограмма могла быть заблокирована до освобождения элемента в очереди. Если такая сопрограмма есть, то будет установлено pdTRUE, иначе - pdFALSE

Возвращает:

pdTRUE, если значение получено, иначе pdFALSE.

6.7.7 vCoRoutineSchedule

```
void vCoRoutineSchedule(void);
```

vCoRoutineSchedule() выполняет сопрограмму имеющую высший приоритет. Сопрограмма будет выполняться до блокировки; либо задача, которая выполняет сопрограмму, будет вытеснена. Сопрограммы используют кооперативную многозадачность только между собой, поэтому

сопрограмма не может быть вытеснена другой сопрограммой, но может быть вытеснена задачей.

Если приложение включает в себя и сопрограммы и задачи, то vCoRoutineSchedule() должна вызываться из idle hook.

7 Список источников

1. http://www.freertos.org/RTOS.html – официальный сайт операционной системы
2. Андрей Курниц "FreeRTOS – операционная система для микроконтроллеров".
3. Richard Barry "Using the FreeRTOS Real Time Kernel – a Practical Guide".
4. http://wiki.fh-up.ru/index.php?title=FreeRTOS – Русская документация на FreeRTOS
5. http://ru.wikipedia.org/wiki/FreeRTOS
6. http://emproj.com/FreeRTOS