**Глава 1.**

**Управление задачами.**

**1.1 Введение и обзор.**

Целью этой главы является дать читателям хорошее понимание:

* Как FreeRTOS распределяет время обработки для каждой задачи приложения.
* Как FreeRTOS выбирает задачу, которая должна быть выполнена в любое заданное время.
* Как относительный приоритет каждой задачи влияет на поведение системы.
* Состояния, в которых может существовать задача.

Читатели должны также приобрести хорошее понимание:

* Как реализовать задачи.
* Как создать один или более экземпляров задач.
* Как использовать параметр задачи.
* Как изменить приоритет задачи, которая была создана прежде.
* Как удалить задачу.
* Как реализовать периодическую обработку.
* Когда выполнится холостая задача и как она может быть использована.

Концепция, представленная в этой главе является фундаментальной для понимания того, как использовать FreeRTOS и как ведут себя приложения FreeRTOS. Таким образом, это наиболее детальная глава в книге.

**1.2 Функции задачи.**

Задачи реализованы как C функции. Единственной особенной вещью является их прототип, который должен возвращать void и принимать в качестве параметра указатель на void. Прототип демонстрируется в листинге 1.

Листинг 1. Прототип функции задачи.

void ATaskFunction(void \*pvParameters);

Каждая задача - это небольшая самостоятельная программа. Она имеет точку входа, обычно всегда работает в бесконечном цикле и не выходит. Структура типовой задачи показана в листинге 2.

FreeRTOS задачам не должно дозволяться возвращаться из их реализующих функций никаким способом - они не должны содержать оператор return и им не должно дозволяться выполнение в обход конца функции. Если задача больше не требуется, она должна быть явно удалена. Это также демонстрируется в листинге 2.

Определение единственной функции задачи может быть использовано для создания любого числа задач - каждая созданная задача является отдельным исполняемым экземпляром с его собственным стеком и его собственной копией автоматических (стековых) переменных, определенных в самой задаче.

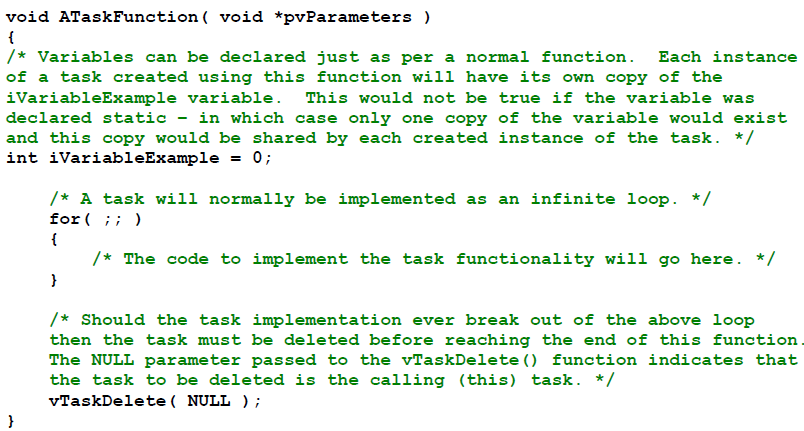
Листинг 2. Структура типовой функции задачи.

Комментарии:

Переменные могут быть объявлены также, как в нормальной функции. Каждый экземпляр задачи, созданный используя эту функцию, будет иметь его собственную копию переменной iVariableExample. Это не было бы истиной, если бы переменная была объявлена статической – в этом случае существовала бы только одна копия переменной и эта копия использовалась бы совместно каждым созданным экземпляром задачи.

Задача будет реализована нормально как бесконечный цикл. В теле цикла размещается код, реализующий функциональность задачи.

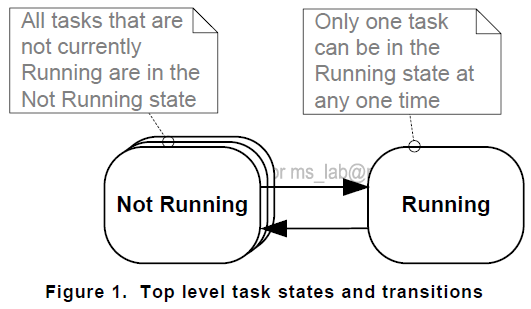
Если реализация задачи когда-либо выходит из вышеупомянутого цикла, тогда задача должна быть удалена прежде, чем будет достигнут конец этой функции. Параметр NULL переданный функции vTaskDelete() показывает, что удаляемая задача является вызывающей эту функцию задачей (эта задача).



**1.3 Состояния задачи верхнего уровня.**

Приложение может состоять из многих задач. Если микроконтроллер, запускающий приложение, имеет одно ядро, то только одна задача может выполняться в любой заданный момент времени. Это подразумевает, что задача может существовать в одном из двух состояний: «Выполняется» и «Не выполняется». Мы будем считать это упрощенной моделью в начале – но имейте ввиду, что это – упрощение, так как мы увидим позже, что состояние «Не выполняется» фактически содержит много под-состояний.

Когда задача находится в состоянии «Выполняется», процессор исполняет ее код. Когда задача в состоянии «Не выполняется», задача бездействует, ее статус сохраняется готовым для возобновления в следующий раз, когда планировщик решит, что она должна войти в состояние «Выполняется». Когда задача возобновляет выполнение, она делает это, начиная с инструкции, которую собиралась выполнить перед тем, как в последний раз оставила состояние «Выполняется».



Когда задача перешла из состояния «Не выполняется» в состояние «Выполняется», то говорят был «переключен в» или «загружен в». Наоборот, когда задача перешла из состояния «Выполняется» в состояние «Не выполняется», то говорят был «выключен» или «выгружен». Планировщик FreeRTOS это только сущность, которая может включать (загружать) задачу и выключать (выгружать).

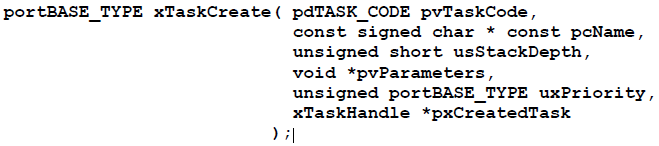
**1.4 Создание задач.**

API функция xTaskCreate().

Задачи создаются использованием API FreeRTOS функции xTaskCreate(). Возможно, она является самой сложной из всех API функций, так что это неудача, что она встретилась нам первой, но задачи должны быть освоены вначале, как самые фундаментальные компоненты многозадачной системы. Все примеры, которые сопровождаются в этой книге, используют функцию xTaskCreate(), так что есть большое количество примеров для опоры.

Раздел 8.5 описывает типы данных и используемые соглашения имен.

Листинг 3. Прототип API функции xTaskCreate().



Далее приведены параметры и возвращаемое значение функции xTaskCreate().

pvTaskCode. Задачи – это простые C функции, которые никогда не выходят и, как таковые, нормально реализованы как бесконечный цикл. Параметр pvTaskCode это простой указатель на функцию (в действительности, просто имя функции), которая реализует задачу.

pcName. Описательное имя задачи. Оно не используется FreeRTOS никаким образом и включено просто как средство отладки. Идентификация задачи по читабельному человеческому названию намного проще, чем пытаться идентифицировать ее по ее хэндлу.

Определенная приложением константа configMAX\_TASK\_NAME\_LEN определяет максималюную длину имени задачи, которое можно взять, включая терминатор NULL. Предоставление строки, размер которой больше, чем этот параметр, приведет к ее отсечке.

usStackDepth. Каждая задача имеет свой собственный уникальный стек, который выделяется ядром задаче, когда она создается. Значение usStackDepth указывает ядру, насколько большим делать стек. Значение определяет количество слов, которые могут храниться в стеке, но не количество байтов. Например, Cortex-M3 стек имеет ширину в 32 бита, так что если в usStackDepth передать 100, то под стек будет выделено 400 байтов. Глубина стека, умноженная на ширину стека не должна превышать максимальное значение, которое может храниться в переменной типа size\_t. Размер стека, используемый холостой задачей, определен в приложении через константу configMINIMAL\_STACK\_SIZE. Значение, назначенное этой константе в стандартных демо-приложениях FreeRTOS Cortex-M3 является минимально рекомендуемым для любой задачи. Если ваша задача использует много пространства стека, то вы должны назначить больший размер.

Нет простого способа определить размер стека, требуемый задаче. Это возможно вычислить, но большинство пользователей просто назначат размер, который по их мнению является разумным значением, затем используют возможности, предоставляемые FreeRTOS, чтобы гарантировать, что выделенное пространство действительно адекватное, и что оперативная память не расходуется впустую. Глава 6 содержит информацию о том, как запросить пространство стека, используемое задачей.

pvParameters. Функции задач принимают параметр типа указателя на void (void\*). Значение, назначенное параметру pvParameters будет значением, переданным задаче. Некоторые примеры в этом документе демонстрируют, как может быть использован параметр.

uxPriority. Определяет приоритет, с которым будет выполняться задача. Приоритет может быть назначен от 0, то есть самый низкий приоритет, до (configMAX\_PRIORITIES - 1) - самый высокий приоритет.

configMAX\_PRIORITIES это определенная пользователем константа. Не существует верхнего предела количества доступных уровней приоритета (кроме предела, определяемого используемыми типами данных и доступной оперативной памяти в вашем контроллере), но вы должны использовать минимально необходимое число уровней приоритета во избежание пустого расхода памяти.

Передача uxPriority значения, превышающему configMAX\_PRIORITIES приведет к тому, что приоритет будет молча ограничен до законного максимального значения.

pxCreatedTask. pxCreatedTask может быть использован для отдачи дескриптора (хэндла) создаваемой задаче. Этот дескриптор может быть использован для ссылки на задачу в API вызовах, которые, например, изменяют приоритет задачи или удаляют задачу.

Если ваше приложение не использует дескрипторы задач, то pxCreatedTask может быть установлен в NULL.

Возвращаемое значение. Возможны два значения:

1. pdTRUE. Это означает, что задача была создана успешно.

2. errCOULD\_NOT\_ALLOCATE\_REQUIRED\_MEMORY. Это означает, что задача не была создана, потому что не хватает памяти в куче, доступной для FreeRTOS, чтобы выделить достаточно RAM для хранения структуры данных задачи и стек.

Глава 5 содержит больше информации об управлении памятью.

Пример 1. Создание задач.

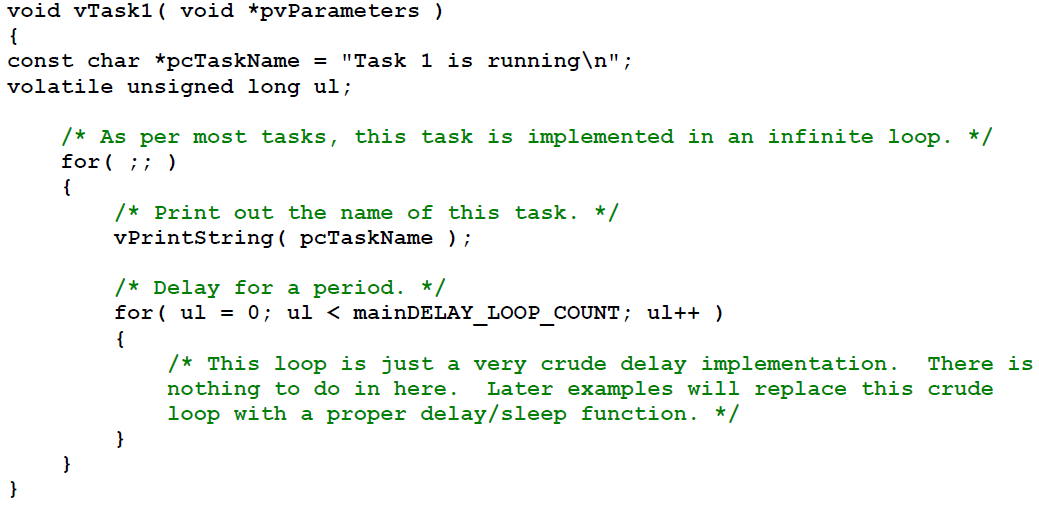
Пример демонстрирует по шагам, как создать две простых задачи, затем запустить выполнение задач. Задачи просто печатают периодически строку, используя сырой пустой цикл для создания временной задержки. Обе задачи созданы с одинаковым приоритетом и являются идентичными, за исключением строки, которую они печатают - см. Листинг 4 и листинг 5 с соответствующими их реализациями.

Комментарии в определении функции void vTask1(void \*pvParameters).

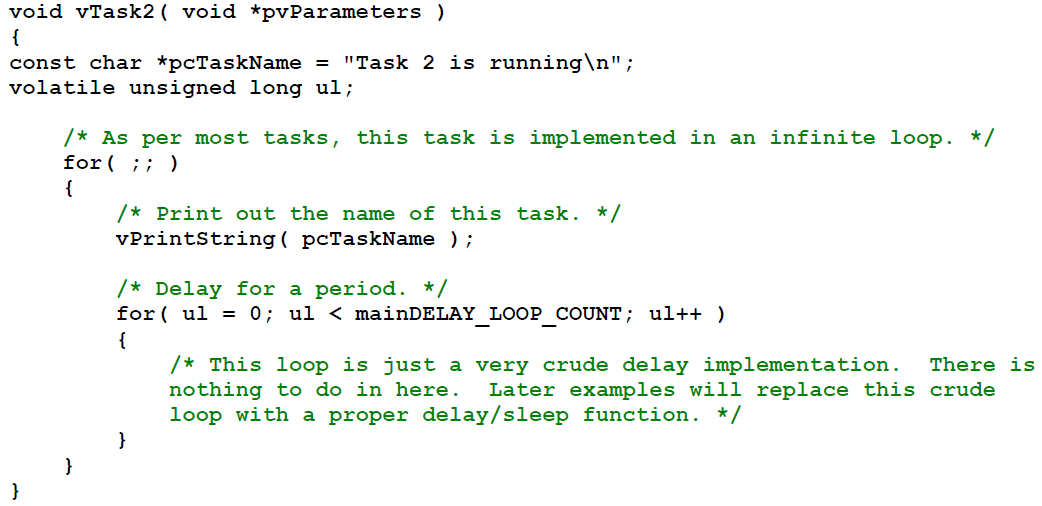
Как и в большинстве задач, эта задача реализована как бесконечный цикл. Данная задача с помощью оператора vPrintString(pcTaskName) выводит на печать имя этой задачи. Затем следует задержка для периода. Далее следует цикл. Этот цикл представляет собой очень простую сырую реализацию. Здесь ничего не делается. Дальнейшие примеры будут заменять этот сырой цикл надлежащей delay/sleep функцией.

Комментарии в определении функции void vTask2(void \*pvParameters). Точно такие же.

Листинг 4. Реализация первой задачи, используемой в примере 1.



Листинг 5. Реализация второй задачи, использованной в примере 1.



Функция main() создает задачи прежде, чем запустится планировщик - см. листинг 6 с ее реализацией.

Комментарии в функции main().

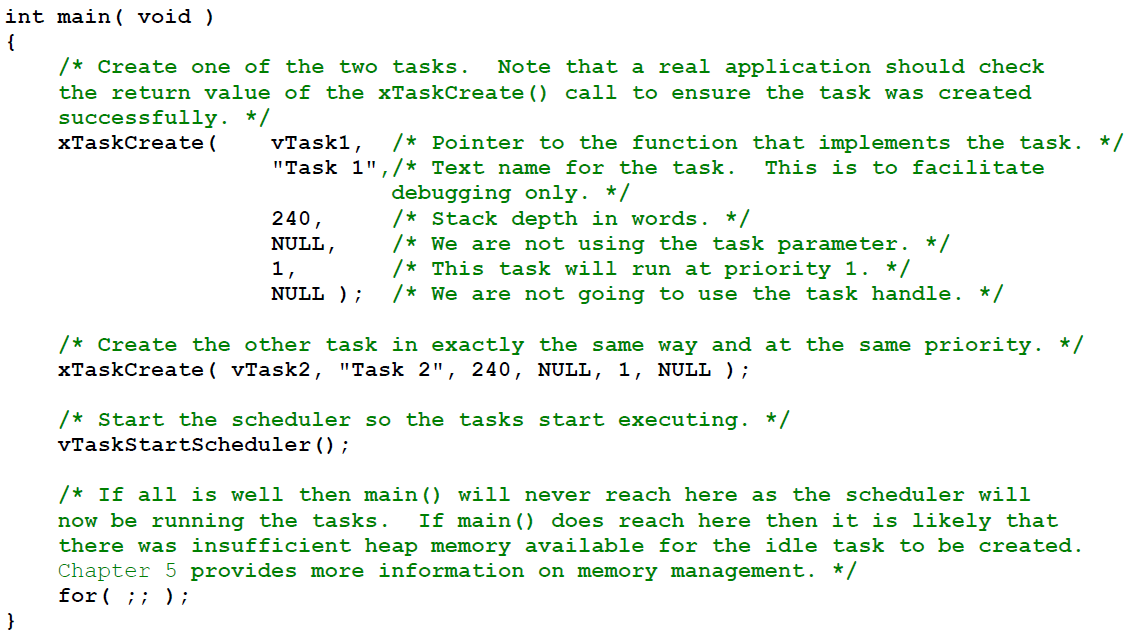
Создание одной из двух задач. Заметьте, что реальное приложение должно проверять возвращаемое значение вызова xTaskCreate(), чтобы убедиться, что задача была создана успешно.

Комментарии аргументов функции xTaskCreate() не привожу, так как смысл каждого ясен из описания в разделе 1.4.

Далее создается другая задача точно таким же образом и с тем же самым приоритетом.

Затем запускается планировщик, так что задачи начинают выполняться.

Листинг 6. Запуск задач примера 1.



Если все хорошо, то функция main() никогда не достигнет оператора for(;;), так как планировщик будет сейчас запускать задачи. Если функция main() достигла этого места, то это будет похоже на то, что произошла нехватка памяти для создания холостой задачи. Глава 5 содержит больше информации об управлении памятью.

Выходная информация, произведенная функцией vPrintString() отображается в выбранной IDE. Выполнение этого примера дает результат, показанный на рисунке 2, скрин которого взят с Red Suite IDE.

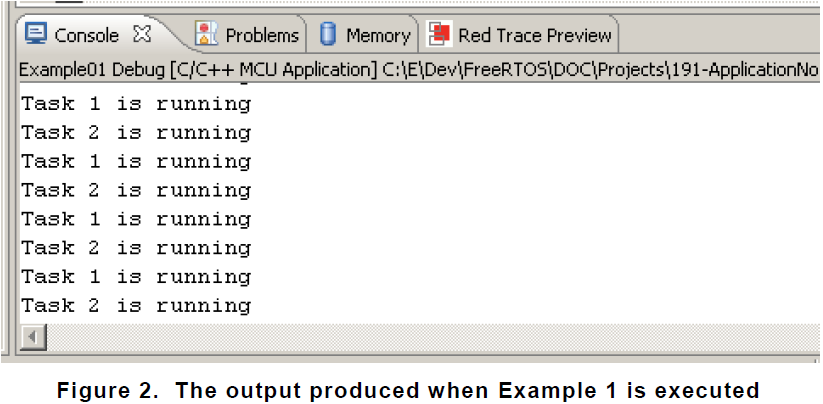
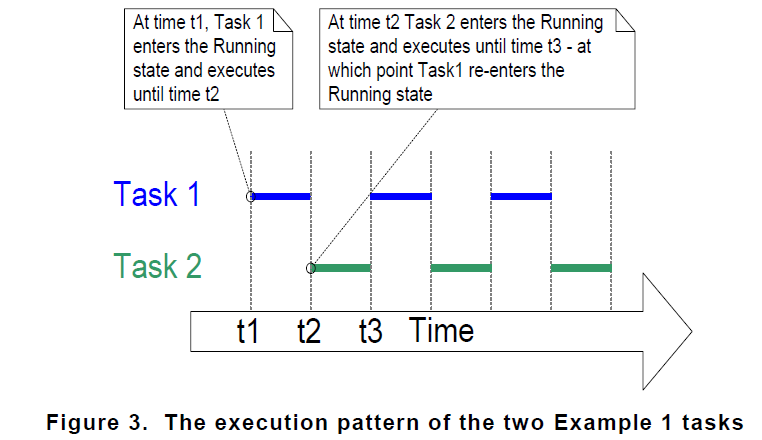


Рисунок 2 показывает, что как будто эти две задачи выполняются одновременно. Однако, так как обе задачи выполняются на одном и том же процессоре, это не может иметь место быть. В действительности, обе задачи быстро входят в состояние "Выполняется" и выходят из него. Обе задачи выполняются с одинковым приоритетом, и поэтому, они разделяют время на одном процессоре. Реальный шаблон их выполнения показан на рисунке 3.

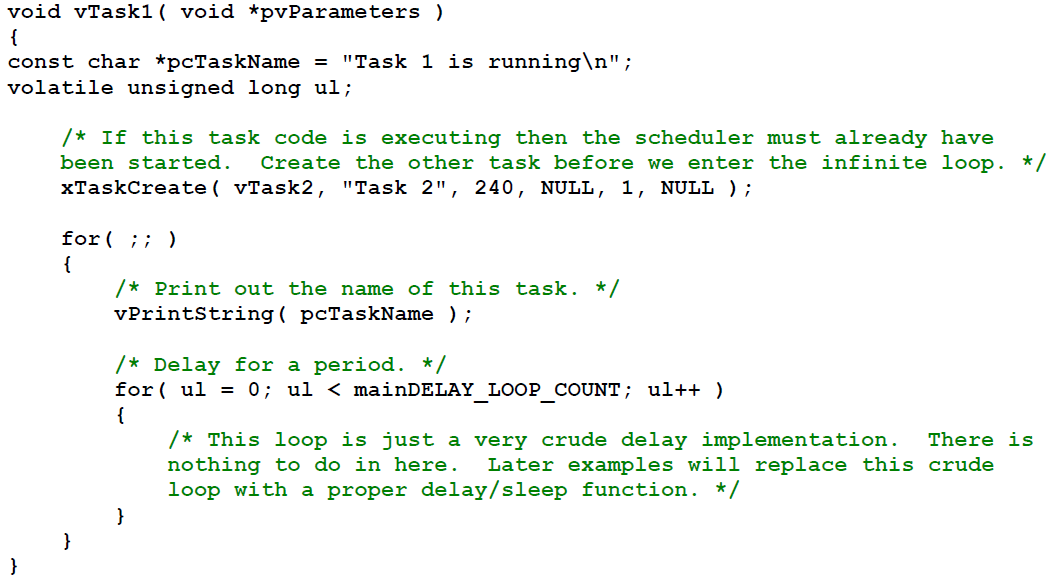
Стрелка вдоль нижней части рисунка 3 показывает течение времени, начиная с момента времени t1 вперед. Цветные линии показывают какая задача выполняется в каждый момент времени. Например, задача 1 выполняется между моментом времени t1 и t2.

Только одна задача может существовать в состоянии "Выполняется" в любой момент времени. Так что, как только одна задача входит в состояние "Выполняется" (задача загружена), другая входит в состояние "Не выполняется" (задача выгружена).



В примере 1 обе задачи созданы внутри функции main() перед запуском планировщика. Возможно также создать задачу изнутри другой задачи. Мы могли бы создать задачу 1 из main(), и затем создать задачу 2 из под задачи 1. Для того, чтобы мы могли сделать это, функция задачи 1 должна измениться, как показано в листинге 7. Задача 2 не создастся до тех пор, пока не будет запущен планировщик, но вывод, произведенный приложением, будет таким же.

Листинг 7. Создание задачи из под другой задачи, после того, как запустится планировщик.

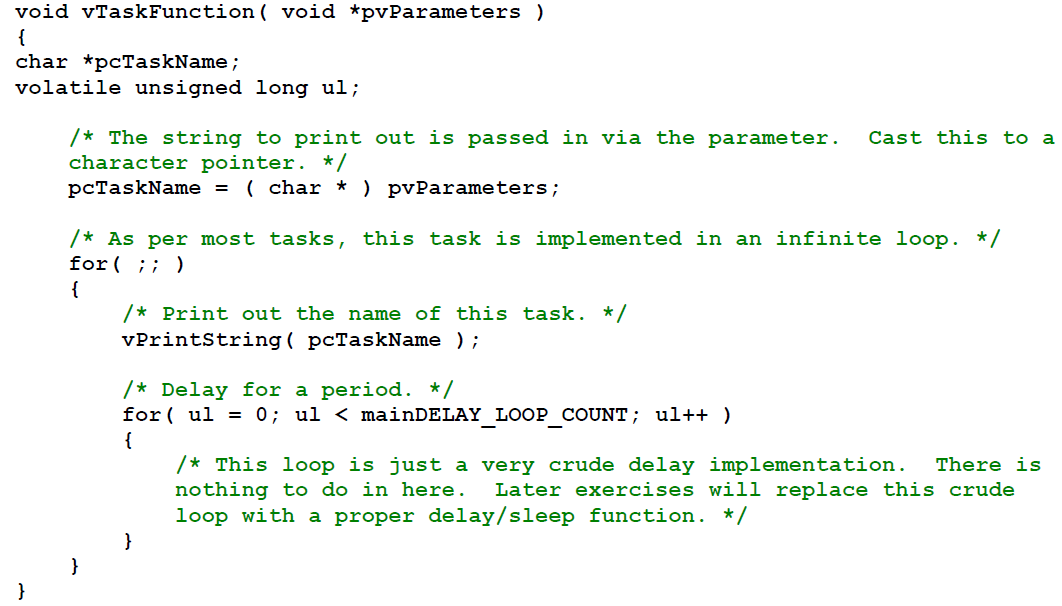


Пример 2. Использование параметра задачи.

Две задачи, созданные в примере 1 почти идентичны, разница есть только между строчками текста, которые они выводят. Такое дублирование может быть удалено, а вместо этого можно создать два экземпляра одной реализации задачи. Тогда может быть использован параметр задачи, который используется для передачи в каждую задачу строку, которую она должна напечатать.

Листинг 8 содержит код единственной функции задачи (vTaskFunction), используемой в примере 2. Эта единственная функция замещает две функции задач vTask1 и vTask2, используемые в примере 1. Заметьте, что параметр задачи приводится к типу (char\*), чтобы получить строку, которую должна напечатать задача.

Листинг 8. Единственная функция задачи, используемая для создания двух задач в примере 2.



Даже при том, что теперь есть только одна реализация задачи, можно создать более одного экземпляра определенной задачи. Каждый созданный экземпляр будет выполняться независимо, под управлением FreeRTOS планировщика.

Параметр pvParameters используется в функции xTaskCreate() для передачи текстовой строки, как показано в листинге 9.

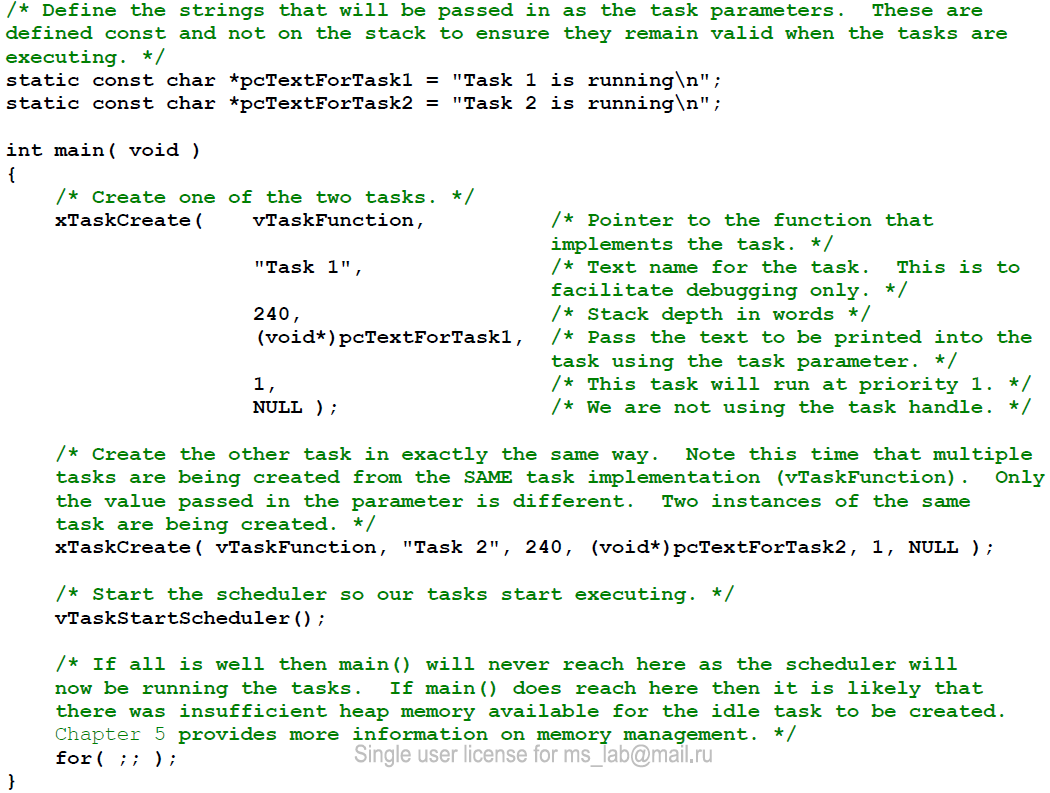
Листинг 9. Функция main() для примера 2.

Комментарии. Вначале идет определение строк, которые будут переданы как параметры задачи. Они определены как константы, и не на стеке, чтобы гарантировать, что они останутся корректными, когда выполняются задачи.

Далее создается одна из двух задач.

Другая задача создается точно таким же образом. Заметьте, что на этот раз несколько задач были созданы из одной и той же реализации задачи (vTaskFunction). Отличие только в значении передаваемого параметра. Два экземпляра одной и тойже задачи были созданы.

Затем запускается планировщик, и, таким образом, задачи начинают выполняться.



Результаты работы примера 2 в точности такие же, как в примере 1, показанные на рисунке 2.

**1.5 Приоритеты задач.**

Параметр uxPriority API функции xTaskCreate() назначает начальный приоритет создаваемой задаче. Приоритет может быть изменен после того как будет запущен планировщик с помощью API функции vTaskPrioritySet().

Максимальное количество доступных уровней приоритета устанавливается определяемой приложением константой конфигурации времени компиляции configMAX\_PRIORITIES внутри файла FreeRTOSConfig.h. Сама FreeRTOS не ограничивает значение этой константы, но более высокое значение константы потребляет больше оперативной памяти, так что всегда рекомендуется устанавливать это значение минимально необходимым.

FreeRTOS не вводит ограничений на то, как приоритеты могут быть присвоены задачам. Любое количество задач может разделять один и тот же приоритет - обеспечивая максимальную гибкость дизайна. Вы можете назначать уникальный приоритет каждой задаче, если желаете (как того требуют некоторые алгоритмы с возможностью планировки), но это ограничение не навязывается никоем образом.

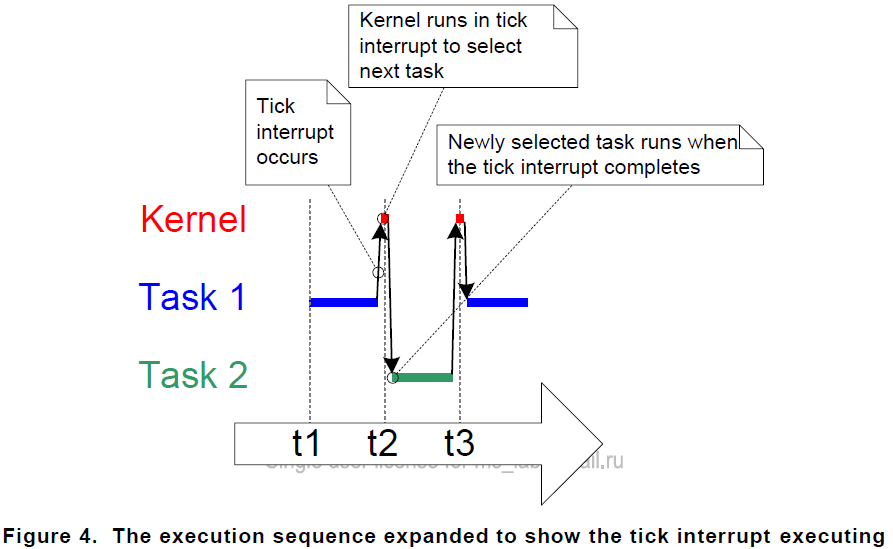
Низкие числовые значения приоритета означают низкоприоритетные задачи, задачи с приоритетом 0 являются самым низким приоритетом. Таким образом, диапазон доступных приоритетов - от 0 до configMAX\_PRIORITIES - 1.

Планировщик всегда гарантирует, что задача с наивысшим приоритетом, которая в состоянии работать, является задачей, выбранной для ввода в состояние "Выполняется". Где более одной задачи, способных к работе, имеющие одинаковый приоритет, там планировщик будет вводит каждую задачу в и из состояния "Выполняется" поочередно. Такое поведение наблюдалось в недавних примерах, где обе задачи создавались с одинаковым приоритетом, и обе всегда были готовы к работе. Каждая такая задача выполняется в течение интервала времени; она входит в состояние "Выполняется" в начале интервала времени, и выходит из этого состояния в конце интервала времени. На рисунке 3, время между t1 и t2 равняется одному временному интервалу.

Чтобы быть в состоянии выбрать следующую задачу для выполнения, сам планировщик должен отработать в конце каждого интервала времени. Периодическое прерывание, называемое тик прерыванием, используется для этой цели. Длительность временного интервала фактически устанавливается частотой тик прерываний, которая настраивается константой конфигурации времени компиляции configTICK\_RATE\_HZ в файле FreeRTOSConfig.h. Например, если configTICK\_RATE\_HZ установить в 100 Гц, тогда временной интервал будет равняться 10 мс. Рисунок 3 может быть расширен, чтобы показать работу самого планировщика в последовательности выполнения. Это показано на рисунке 4.

Заметьте, что FreeRTOS API вызовы всегда определяют время в прерываниях тиков (обычно называемых "тиками"). Чтобы перевести тики в миллисекунды предоставляется специальная константа portTICK\_RATE\_MS. Доступное разрешение зависит от частоты тиков.

Значение "счетчика тиков" - это общее количество прерываний тиков, которые произошли с тех пор как был запущен планировщик, с оговоркой, что счетчик тиков не был переполнен. Пользовательские приложения не должны рассматривать переполнения, когда определяют временные задержки, так как временная непротиворечивость управляется ядром.

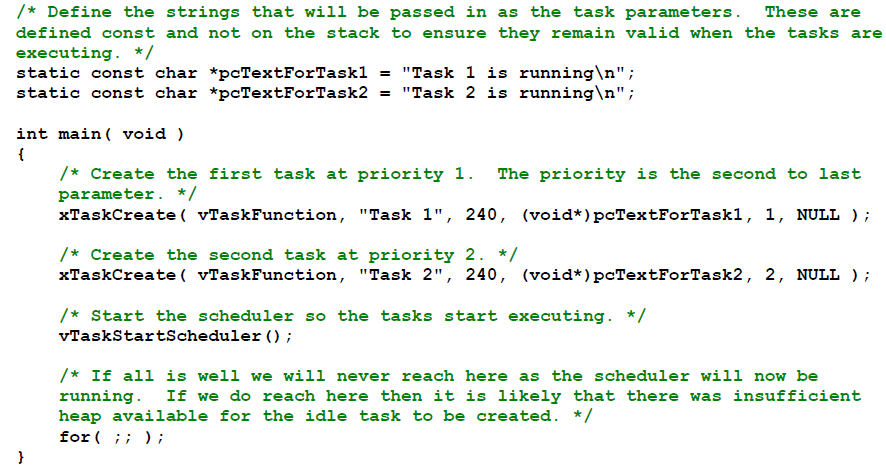


На рисунке 4 верхними короткими линиями (красненькие) показаны моменты, когда работает само ядро. Стрелочки показывают последовательность выполнения от задачи до прерывания, затем от прерывания обратно к другой задаче.

Пример 3. Эксперименты с приоритетами.

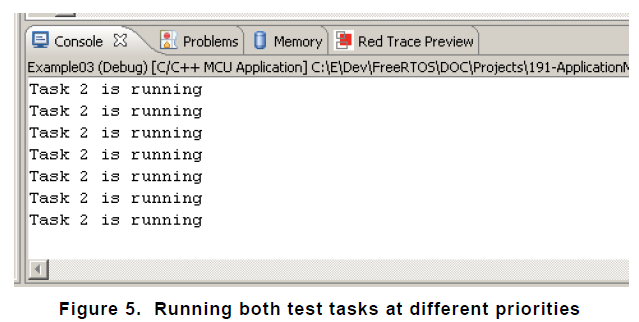
Планировщик всегда гарантирует, что задача с наиболее высоким приоритетом, готовая к работе, будет выбрана для введения в состояние «Выполняется». До сих пор в наших экспериментах создавались две задачи с одинаковым приоритетом, так что обе вводились и выводились в с/из состояние «Выполняется» поочередно. Этот пример показывает, что случится, когда мы изменим приоритет одной из двух задач, созданной в примере 2. На этот раз, первая задача будет создана с приоритетом 1, а вторая задача с приоритетом 2. Код для создания задач показан в листинге 10. Единственная функция, которая реализует обе задачи не изменится, она все еще просто печатает периодически строку, используя пустой цикл для создания задержки.

Листинг 10. Создание двух задач с разными приоритетами.



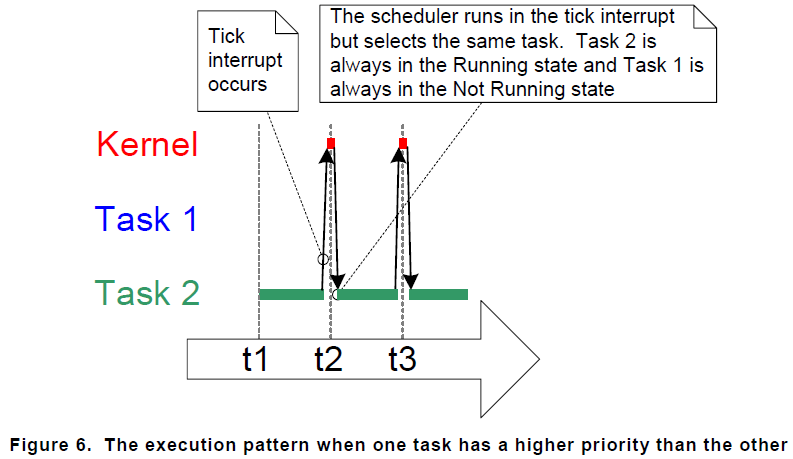
Вывод, произведенный примером 3 показан на рисунке 5.

Планировщик всегда выбирает для запуска готовую к работе задачу с наивысшим приоритетом. Задача 2 имеет наивысший приоритет, по сравнению с задачей 1 и она всегда готова к работе. Таким образом, только задача 2 всегда входит в состояние «Выполняется». Так как задача 1 никогда не входит в состояние «Выполняется», то она никогда не выведет на печать ее строку. Задача 1, как говорят, «черпает ресурс» времени обработки задачей 2.



Задача 2 всегда готова к выполнению, поскольку она никогда ничего не ждет. Она либо циклично крутится в цикле, либо печатает в терминал.

Рисунок 6 показывает последовательность выполнения для примера 3.



Пояснение: планировщик запускается в тик прерываниях, но выбирает одну и туже задачу. Задача 2 всегда в состоянии «Выполняется», а задача 1 всегда в состоянии «Не выполняется».

**1.6 Расширение понятия состояния "Не выполняется".**

До сих пор, создаваемые задачи всегда были готовыми к работе и никогда ничего не должны были ожидать, и поскольку они ничего не должны были ждать, они всегда были готовы для входа в состояние "Выполняется". Такой тип "непрерывных" задач является ограничено полезным, потому что они могут быть созданы только с самым низким приоритетом. Если они запускаются на любом другом приоритете, они предотвращают задачи с более низким приоритетом запущенные когда либо вообще.

Чтобы сделать наши задачи полезными, нам нужен способ позволить им быть управляемыми событиями. Управляемая событием задача содержит работу (обработку) для выполнения ее только после того, как произойдет событие, которое инициировало задачу. Планировщик всегда выбирает задачу с сымым высоким приоритетом, которая готова к запуску. Высокоприоритетные задачи, не готовые к запуску, подразумевают, что планировщик не может выбрать их и должны вместо этого выбрать задачу с менее высоким приоритетом, которая готова к запуску. Таким образом, использование событийно-ориентированных задач предполагает, что задачи могут быть созданы с различными приоритетами, без задач с наивысшим приоритетом, захватывающих весь ресурс времени обработки задач с менее высоким приоритетом.

**Состояние "Блокирован".**

Говорят, что задача, которая ожидает события, находится в "Блокированном" состоянии, которое является подсостоянием состояния "Не выполняется".

Задачи могут входить в состояние "Блокирован" для ожидания двух различных типов событий:

1. Временные события (time-related) - это события, которые либо представляют собой истечение периода задержки, или достигнута абсолютная величина времени. Например, задача может войти в состояние "Блокирован" для ожидания происшествия 10 мс времени.
2. События синхронизации - где события пораждаются из другой задачи или прерывания. Например, задача может войти в состояние "Блокирован" для ожидания данных, прибывающих в очередь. События синхронизации охватывают широкий спектр типов событий.

FreeRTOS очереди, двоичные семафоры, счетные семафоры, рекурсивные семафоры и мьютексы могут быть использованы для создания событий синхронизации. Глава 2 и глава 3 охватывают эти моменты более детально.

Возможно, что задача блокируется в ожидании события синхронизации с таймаутом, т.е. фактически она блокируется на оба типа событий одновременно. Например, задача может выбрать ожидание втечение 10 мс прибывания данных в очередь. Задача покинет состояние "Блокирован" либо если данные придут втечение 10 мс, либо если 10 мс истекают без приема данных.

**Состояние "Приостановлен".**

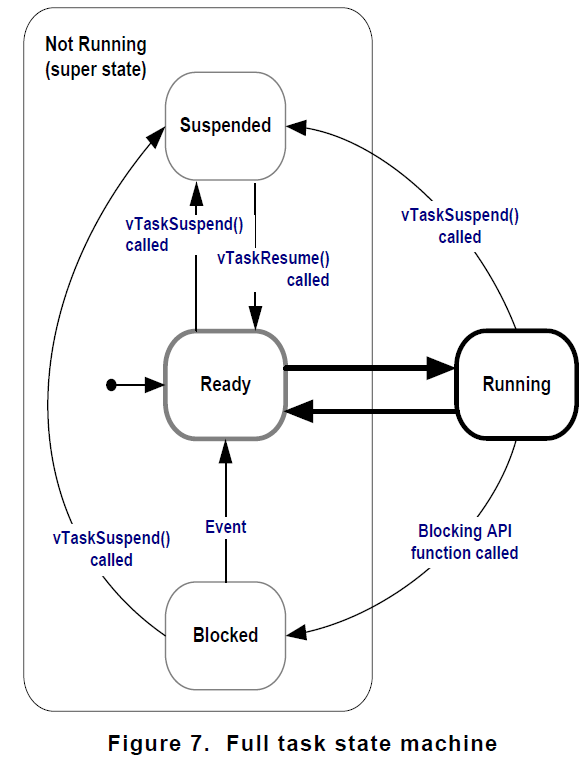
"Приостановлен" - это также подсостояние состояния "Не выполняется". Задачи в состоянии "Приостановлен" не доступны планировщику. Единственный способ которым можно ввести задачу в состояние "Приостановлен" заключается в вызове API функции vTaskSuspend(). Единственный способ вывести задачу из этого состояния заключается в вызове API функции vTaskResume() или xTaskResumeFromISR(). Большинство приложений не используют состояние "Приостановлен".

**Состояние "Готов".**

Говорят, что задачи, которые находятся в состоянии "Не выполняется", но не являются блокированными, или приостановленными, находятся в состоянии "Готов". Они способны работать, и таким образом, готовы выполняться, но пока еще не находящиеся в состоянии "Выполняется".

Завершение диаграммы переходов состояний.

Рисунок 7 расширяет предыдущую очень упрощенную диаграмму состояний с целью включить все подсостояния состояния "Не выполняется", описанные в этом разделе. Задачи, создаваемые в примерах, до сих пор не использовали состояния "Блокирован" и "Приостановлен", они только взаимопереходили в состояния "Готов" и "Выполняется" - выделено жирными линиями на рисунке 7.



Все задачи, создаваемые в представленных до сих пор примерах были "периодическими" - они задерживались на величину периода и выводили на печать их строчки, перед тем, как задержаться еще раз, и .т.д. Задержки были реализованы очень "сыро", используя пустой цикл - фактически опрашивается инкрементирующий счетчик цикла до тех пор, пока он не достигнет определенного значения. Пример 3 ясно демонстрирует недостатки этого метода. Пока выполняется пустой цикл, задача остается в состоянии "Готов", затрачивая ресурсы времени обработки другой задачи.

Существует несколько других недостатков в любых формах опроса, не в последнюю очередь один из которых - неэффективность. Во время опроса задача на самом деле не выполняет никакой работы, но все еще использует максимум процессорного времени и таким образом, тратит процессорные циклы. Пример 4 корректирует это поведение, замещая пустой цикл опроса вызовом API функции vTaskDelay(), прототип которой приведен в листинге 11. Определение новой задачи показано в листинге 12. Заметьте, что API функция vTaskDelay() доступна только, когда INCLUDE\_vTaskDelay установлен в 1 в файле FreeRTOSConfig.h.

vTaskDelay() помещает вызывающую функцию в состояние "Блокирован" на фиксированное число тиков. Пока задача находится в этом состоянии, она не использует процессорное время, так что процессорное время потребляется только когда есть работа для выполнения.

Листинг 11. Прототип API функции vTaskDelay().

void vTaskDelay(portTickType xTicksToDelay);

Далее приведены параметры и возвращаемое значение функции.

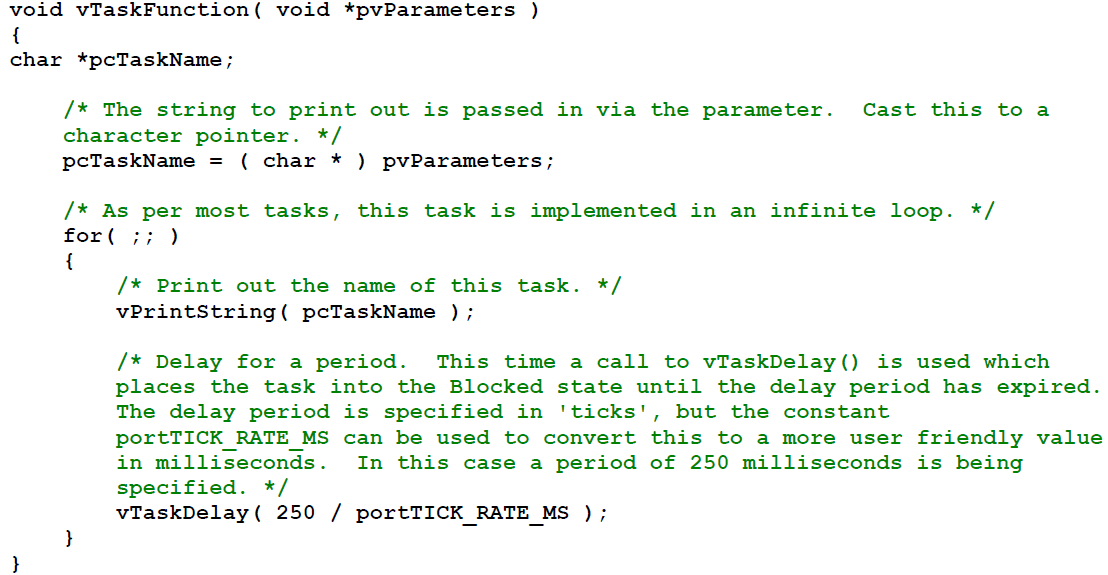
xTicksToDelay - число прерываний тиков, втечение которых, задача должна оставаться в состоянии "Блокирован", прежде чем она переидет обратно в состояние "Готов".

Например, если задача вызывает vTaskDelay(100), в то время как счетчик тиков равен 10000, тогда она должна немедленно войти в состояние "Блокирован" и оставаться в нем до тех пор, пока счетчик типов не достигнет 10100.

Константа portTICK\_RATE\_MS может быть использована для преобразования миллисекунд в тики.

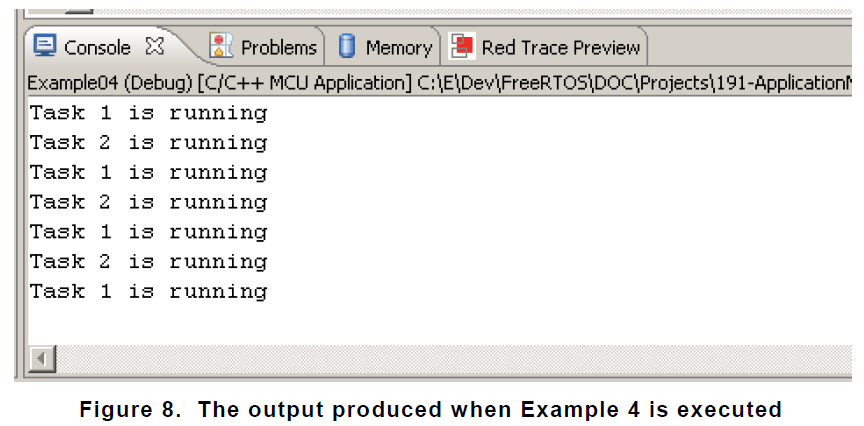
Листинг 12. Исходный код примера, в котором пустой цикл задачи замещен вызовом vTaskDelay().

Комментарий перед вызовом xTaskDelay(). Задержка для периода. На этот раз используется вызов vTaskDelay(), который помещает задачу в состояние "Блокирован" до тех пор, пока не истечет период задержки. Период задержки определяется в "тиках", но константа portTICK\_RATE\_MS может быть использована для преобразования тиков в более дружественное для пользователя значение в миллисекундах. В данном слечае определен период в 250 мс.

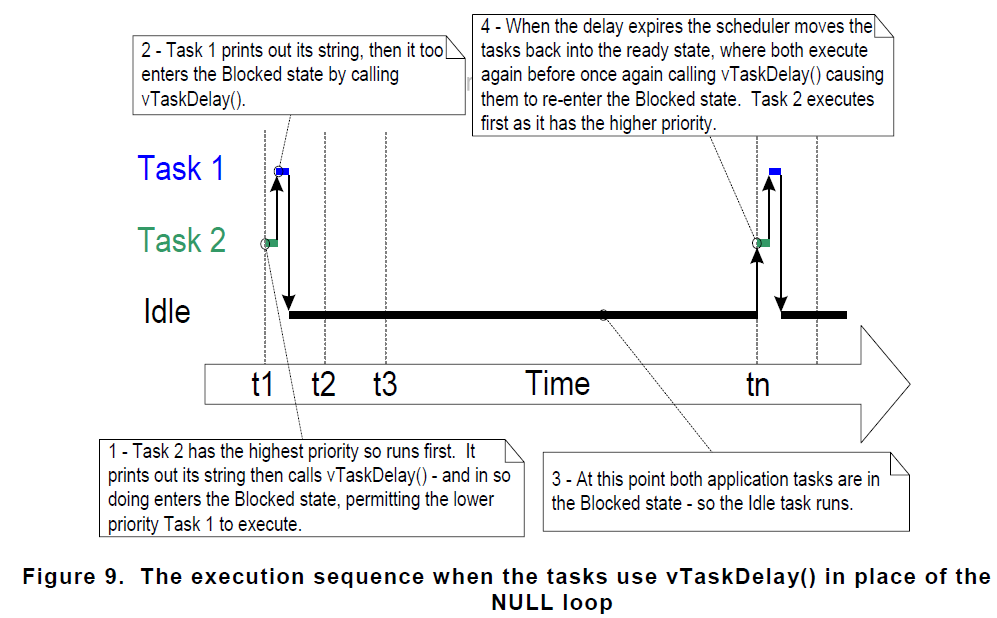


Даже при том, что две задачи все еще создаются с разными приоритетами, сейчас будут работать обе. Вывод примера 4 показан на рисунке 8, который подтверждает ожидаемое поведение.

Последовательность выполнения, показанная на рисунке 9 объясняет, почему запускаются обе задачи, несмотря на то, что они создаются с различными приоритетами. Выполнение самого ядра опущено для упрощения.



Холостая задача создается автоматически, когда запускается планировщик, для гарантии, что всегда есть как минимум одна задача, которая готова к работе (по крайней мере одна задача в состоянии "Готов"). Раздел 1.7 описывает холостую задачу более детально.



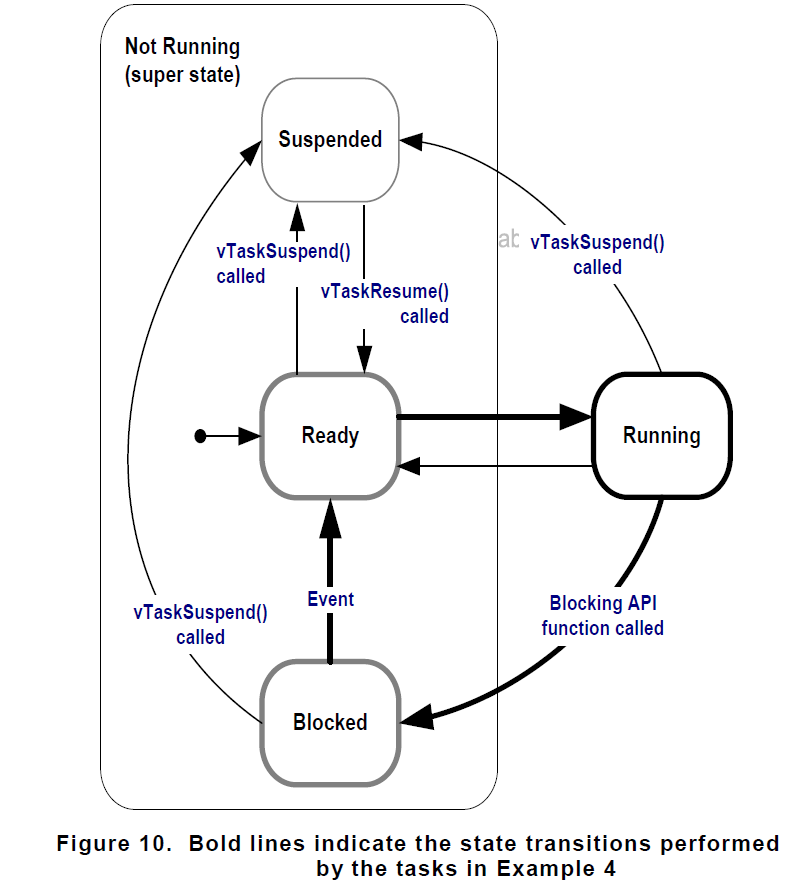
1. Задача 2 имеет наивысший приоритет, так что она запускается первой. Задача 2 выводит на печать ее строчку, затем вызывает vTaskDelay() и при этом входит в состояние "Блокирован", позволяя выполняться задаче 1 с менее высоким приоритетом.
2. Задача 1 выводит на печать ее строчку, затем тоже входит в состояние "Блокирован" посредством вызова vTaskDelay().
3. На этом этапе обе задачи приложения блокированы, так что работает холостая задача.
4. Когда задержка истекла, планировщик переводит задачи обратно в состояние "Готов", и обе выполняются снова до очередного вызова vTaskDelay(), заставляющего вновь войти их в состояние "Блокирован". Задача 2 выполняется первой, так как имеет более высокий приоритет.

Изменена только реализация наших двух задач, но не их функциональность. Сравнение рисунка 9 с рисунком 4 ясно демонстрирует, что эта функциональность была достигнута более эффективным образом.

Рисунок 4 показывает шаблон выполнения, когда задачи используют пустой цикл для создания задержки - так что они всегда готовы к работе и в результате используют много процессорного времени. Рисунок 9 показывает паттерн выполнения, когда задачи входят в блокированный режим на все время их периода задержки, так что процессорное время используется только когда они реально имеют работу, которую нужно выполнить (в данном случае печатается простое сообщение).

По сценарию рисунка 9, всякий раз, когда задачи покидают состояние блокирования, они выполняются втечение части тик-периода, перед тем, как заблокируются снова. Большую часть времени в приложении не существует задач, готовых к запуску (нет задач приложения в состоянии "Готов") и, таким образом, нет задач приложения, которые могут быть выбраны для ввода их в состояние "Выполняется". Пока дело обстоит так, будет выполняться холостая задача. Суммарное время обработки, которое получает неактивная задача, является мерой запасного объема процессорного времени в системе.

Жирные линии на рисунке 10 показывают переходы, выполняемые задачами в примере 4. На этот раз каждый переход осуществляется через состояние "Блокирован", перед тем, как задача снова войдет в состояние "Готов".



**API функция vTaskDelayUntil().**

vTaskDelayUntil() похожа на vTaskDelay(). Как уже было продемонстрировано, параметр vTaskDelay() определяет число прерываний тиков, которые должны произойти между вызовом задачей vTaskDelay() и моментом, когда эта же самая задача выйдет из состояния "Блокирован". Количество времени, втечение которого задача остается блокированной, определяется параметром vTaskDelay(), но реальное время, когда задача покидает состояние "Блокирован", отсчитывается относительно времени, когда была вызвана vTaskDelay().

Вместо этого, параметр vTaskDelayUntil() определяет точное значение числа тиков, на котором вызывающая функция должна будет перейти из состояния "Блокирован" в состояние "Готов". vTaskDelayUntill() это API функция, которая должна быть использована, когда требуется фиксированный период исполнения (где вы хотите, чтобы ваша задача выполнялась периодически, с фиксированной частотой), так время, при котором вызывающая функция разблокируется, является абсолютным, что предпочтительнее, чем относительно момента, когда функция была вызвана (как в случае с vTaskDelay()). Заметьте, что API функция vTaskDelayUntil() доступна только, когда INCLUDE\_vTaskDelayUntil установлен в 1 в файле FreeRTOSConfig.h.

Листинг 13. Прототип API функции vTaskDelayUntil().

void vTaskDelayUntil( portTickType\* pxPreviousWakeTime,

portTickType xTimeIncrement);

Далее приведены параметры и возвращаемое значение функции.

pxPreviousWakeTime - этот параметр называют при условии, что функция vTaskDelayUntil() используется для реализации задачи, которая выполняется периодически и с фиксированной частотой. В этом случае pxPreviousWakeTime хранит время, в которое задача в последний раз покинула состояние "Блокирован" (была разбужена). Это время используется как опорная точка для вычисления времени, в которое задача должна будет покинуть состояние "Блокирован" в следующий раз.

Переменная, на которую указывает pxPreviousWakeTime обновляется автоматически внутри функции vTaskDelayUntil(). Обычно эта переменная не изменяется кодом приложения, кроме того случая, когда переменная инициализируется впервые. Листинг 14 демонстрирует, как выполняется инициализация.

xTimeIncrement - этот параметр также называется при условии, когда vTaskDelayUntil() используется для реализации задачи, которая выполняется периодически и с фиксированной частотой - частота устанавливается значением xTimeIncrement.

xTimeIncrement определяется в тиках. Константа portTICK\_RATE\_MS может быть использована для преобразования миллисекунд в тики.

***NB:*** Эта функция, говоря простыми словами, не учитывает время, затрачиваемое на работу задачи. Перед ее вызовом мы делаем "засечку" времени, сразу как только функция начнет работать. В конце работы задачи вызывается xTaskDelayUntil(<засечка>, <интервал>), которая блокирует функцию. Согласно моим представлениям, задача разблокируется в момент времени (<засечка> + <интервал>). Т.е. период точно детерминирован. Кстати засечку мы делаем сами только один раз - в первый раз. Потом засечка обновляется автоматически. В случае использования vTaskDelay() время блокировки отсчитывается с момента вызова функции блокировки. Т.е. если используется vTaskDelay(<интервал>), то период работы задачи будет (<работа> + <интервал>). <работа> - время работы задачи не может быть строго детерминировано, следовательно период работы задачи также не является детерминированным.

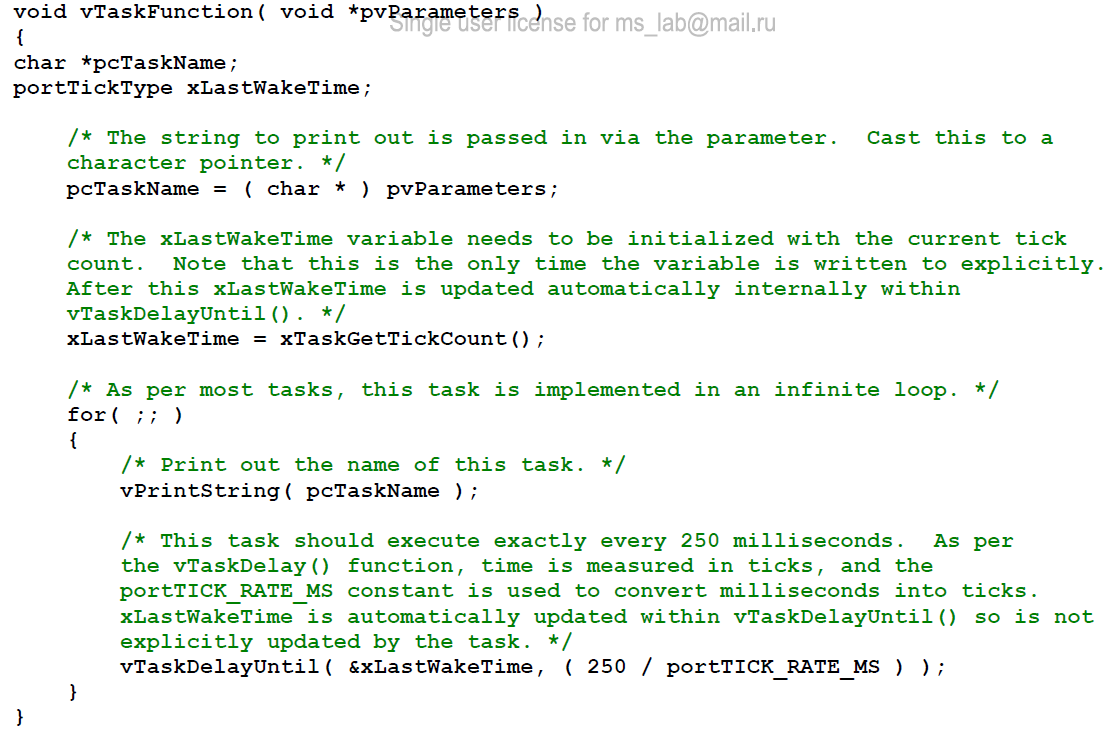
Пример 5. Преобразование задач примера для использования функции vTaskDelayUntil().

Две задачи, созданные в примере 4, это периодические задачи, но использование vTaskDelay() не гарантирует, что частота на которой они работают - фиксирована, так как время, в которое задачи покидают блокированное состояние отсчитывается с момента вызова vTaskDelay(). Преобразуя задачи для использования vTaskDelayUntil() вместо vTaskDelay() решает эту потенциальную проблему.

Листинг 14. Реализация задачи примера, используя vTaskDelayUntil().

Комментарии. Переменная xLastWakeTime должна быть инициализирована значением текущего числа тиков. Заметьте, что это единственный случай, когда переменная была записана явно. После этого xLastWakeTime обновляется автоматически внутри функции vTaskDelayUntil().

Эта задача должна выполняться точно каждые 250 мс. Как и в vTaskDelay() функции, время измеряется в тиках. xLastWakeTime автоматически обновляется внутри функции vTaskDelayUntil() так что она обновляется задачей неявно.



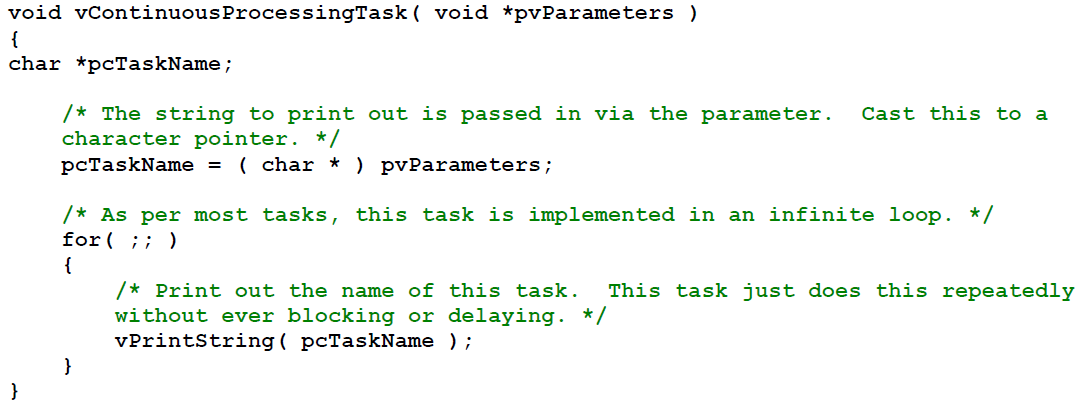
Вывод, произведенный примером 5 точно такой же, что и на рисунке 8 для примера 4.

Пример 6. Комбинирование блокировочной и не блокировочной задач.

Предыдущие примеры исследовали поведение обоих типов задач - опрашивающих и блокирующих в отдельности. Этот пример закрепляет ожидаемое поведение системы путем демонстрации последовательности выполнения, когда две схемы скомбинированы следующим образом:

1. Две задачи создаются с приоритетом 1. Они ничего не делают кроме того, что непрерывно печатают строку. Эти задачи никогда не вызывают никакие API функции, которые могут заставить их войти в состояние "Заблокирован", так что они всегда находятся либо в состоянии "Готов", либо в состоянии "Выполняется". Задачи такой природы называются задачами "непрерывной обработки", так как у них всегда есть работа для выполнения (хотя в данном случае тривиальная работа). Исходный код задач непрерывной обработки показан в листинге 15.
2. Третья задача затем создается с приоритетом 2. Эта задача имеет приоритет выше чем другие две задачи. Третья задача также просто выводит на печать строку, но в этот раз периодически, так как использует API функцию vTaskDelayUntil() для помещения себя самой в состояние "Блокирован" после каждой итерации печати. Исходный код для периодической задачи показан в листинге 16.

Листинг 15. Задача непрерывной обработки, используемая в примере 6.



Листинг 16. Периодическая задача в примере 6.

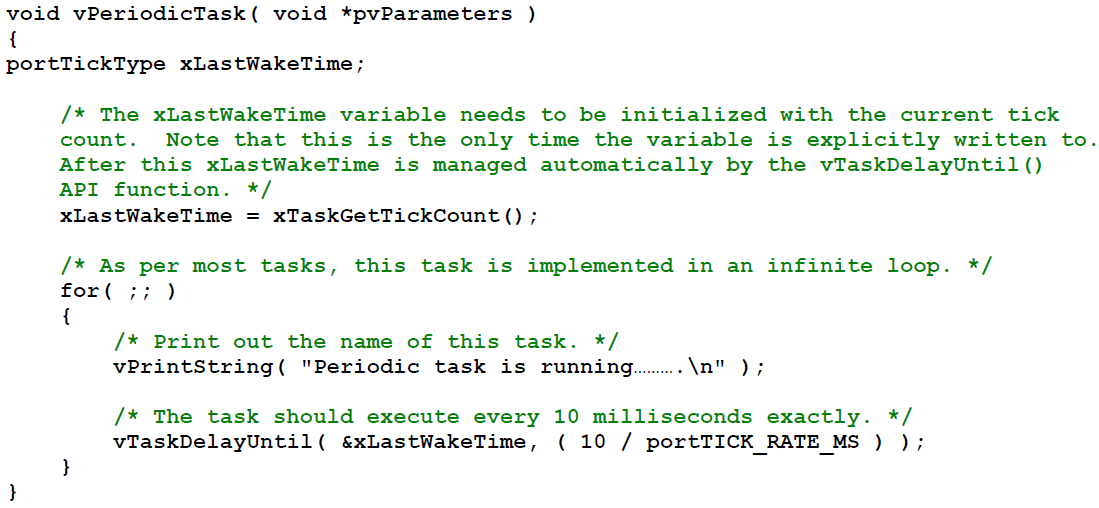
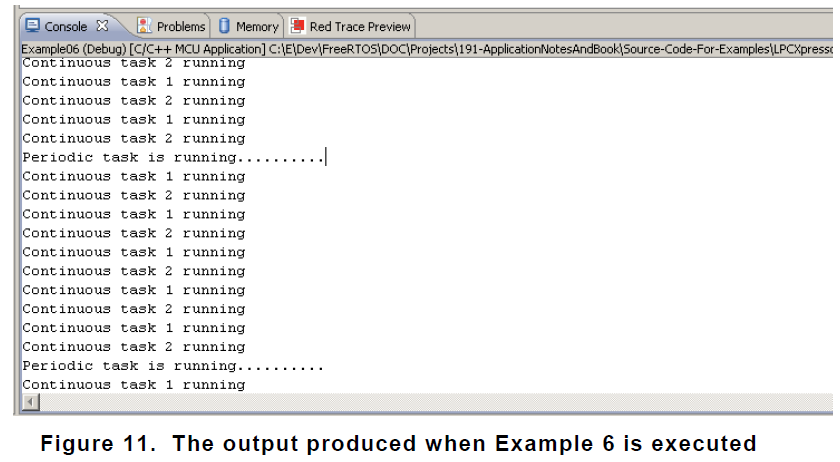
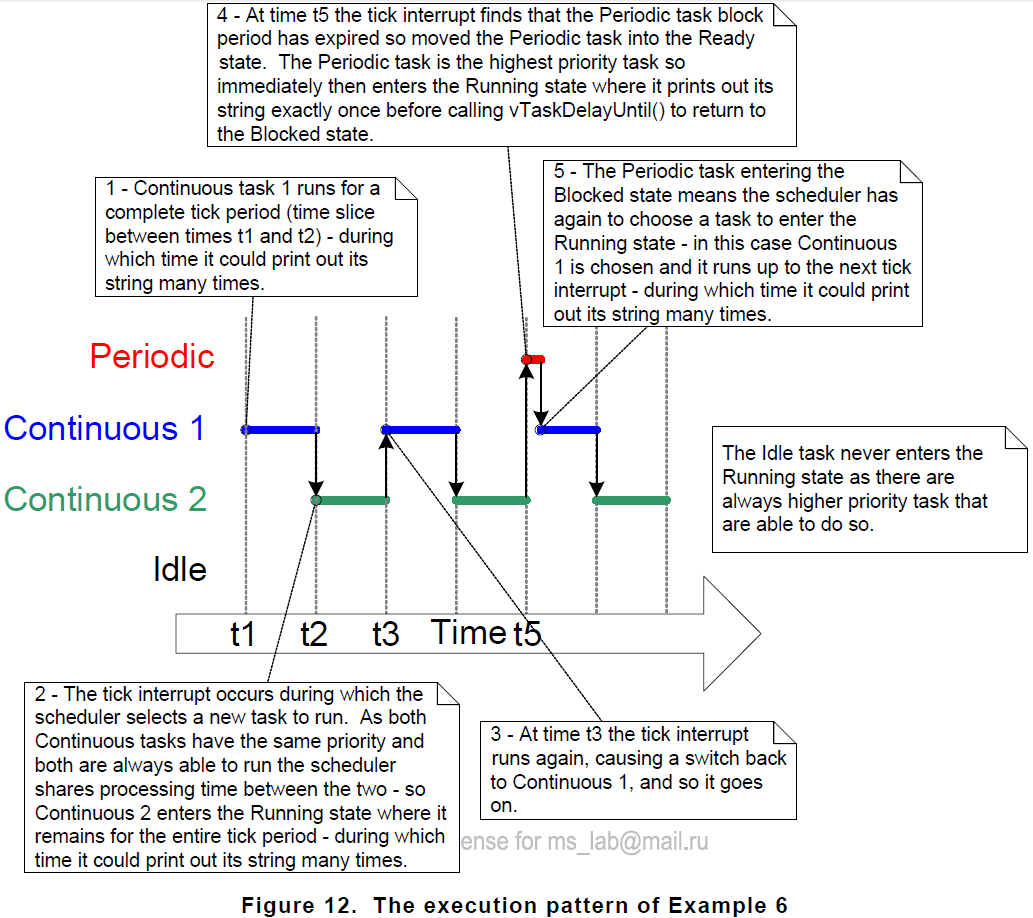


Рисунок 11 показывает вывод, произведенный примером 6. Объяснение наблюдаемого поведения, данного последовательностью выполнения, показано на рисунке 12.



1. Непрерывная задача 1 работает втечение 1 тик периода (временной интервал между моментами времени t1 и t2) - в это время она может выводить на печать ее строку много раз.
2. Происходит тик прерывание, во время которого планировщик выбирает новую задачу для запуска. Так как обе непрерывные задачи имеют одинаковый приоритет и обе всегда готовы к работе, планировщик разделяет процессорное время между двумя задачами - так непрерывная задача 2 входит в состояние "Выполняется", где она и остается на весь тик период, втечение которого она может выводить на печать ее строку много раз.
3. В момент времени t3 тик прерывание срабатывает снова, заставляя включиться первую задачу и так продолжается далее.
4. В момент времени t5 тик прерывание обнаруживает, что период блокировки задачи истек и поэтому периодическая задача помещается в состояние "Готов". Периодическая задача - это задача с наивысшим приоритетом, так что она немедленно входит в состояние "Выполняется", в котором она выводит на печать ее строку только раз, перед тем как вызов vTaskDelayUntil() вернет ее в состояние "Блокирован".
5. Периодическая задача, входящая в блокированное состояние, означает, что планировщик должен снова выбрать задачу для ввода ее в состояние выполнения. В этом случае выбирается непрерывная задача 1, которая продолжает работать до следующего тик прерывания. За это время она может выводить на печать ее строку много раз.

Холостая задача никогда не входит в состояние "Выполняется", так как есть всегда более приоритетные задачи, которые способны делать это.



**1.7 Холостая задача и обработчик холостой задачи.**

Задачи, созданные в примере 4 проводят большую часть их времени в блокированном состоянии. Пока они блокированы, они не способны выполняться и не могут быть выбраны планировщиком.

Процессору всегда нужно что то делать - всегда должна быть по крайней мере одна задача, которая может войти в состояние "Выполняется". Для гарантии этого, холостая задача автоматически создается планировщиком, когда вызывается vTaskStartScheduler(). Холостая задача делает чуть больше, чем сидит в цикле, так что подобно задачам в первых примерах, она всегда готова к работе.

Холостая задача имеет минимально возможный приоритет (0), чтобы гарантировать, что она никогда не предотвратит задачу приложения с более высоким приоритетом от входа в состояние "Выполняется".

Работа на самом низком приоритете гаранетирует, что холостая задача немедленно выйдет из состояния "Выполняется" сразу после того, как более высокоприоритетная задача войдет в состояние "Готов". Этот момент показан на рисунке 9 в момент tn, когда холостая задача немедленно выгрузилась, позволяя задаче 2 выполняться сразу после того, как она покинула состояние "Блокирован". Говорят, задача 2 вытеснила холостую задачу. Вытеснение происходит автоматически, и без знаний о вытесняемой задачи.

**Функции обработчика холостой задачи.**

Возможно добавить в приложение специфическую функциональность прямо в холостую задачу посредством использования обработчика холостой задачи (или callback функции холостой задачи) - функции, которая вызывается автоматически холостой задачей единожды за итерацию цикла холостой задачи.

Общие применения обработчика холостой задачи включают:

* Выполнение низкоприоритетной, фоновой, или непрерывной обработки.
* Измерение количества запасного процессорного времени. (Холостая задача запустится, только когда другие задачи не имеют работы для выполнения, таким образом измеряя количество процессорного времени, выделенного холостой задаче, обеспечивая ясную индикацию запасного процессорного времени.
* Перевод процессора в режим низкого энергопотребления, обеспечивающий автоматический метод сохранения мощности всякий раз, когда не выполняется обработка приложения.

**Ограничения на реализацию функций обработчиков холостых задач.**

Функции обработчиков холостых задач должны придерживаться следующих правил:

1. Функция обработчика холостой задачи не должна никогда пытаться блокировать или приостанавливать. Холостая задача будет выполняться только когда никакие другие задачи не способны делать это (за исключением задач приложения, которые разделяют приоритет холостой задачи).   
   *Примечание:* блокирование холостой задачи любым способом может привести к тому, что не будет задач, которые могут войти в состояние выполнения.
2. Если приложение использует vTaskDelete() API функцию, тогда обработчик холостой задачи должен всегда возвращать управление вызывающей функции в разумных пределах времени. Это нужно потому, что холостая задача ответственна за очистку ресурсов ядра, после того, как задача была удалена. Если холостая задача остается постоянно в функции обработчика холостой задачи, то очистка не может выполняться.

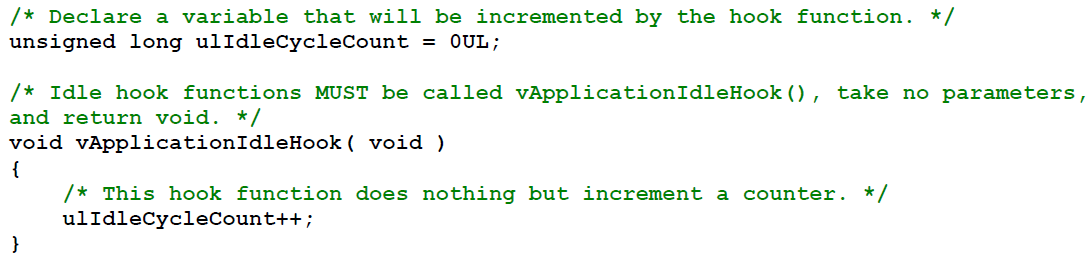
Функции обработчиков холостых задач должны иметь имя и прототип, показанные в листинге 17.

void vApplicationIdleHook(void);

Пример 7. Определение функции обработчика холостой задачи.

Использование блокирующих API вызовов vTaskDelay() в примере 4 создает много времени простоя - времени, когда выполняется холостая задача, потому что обе задачи приложения находятся в состоянии "Блокирован". Пример 7 использует это время посредством добавления функции обработчика холостой задачи, исходный код которой приведен в листинге 18.

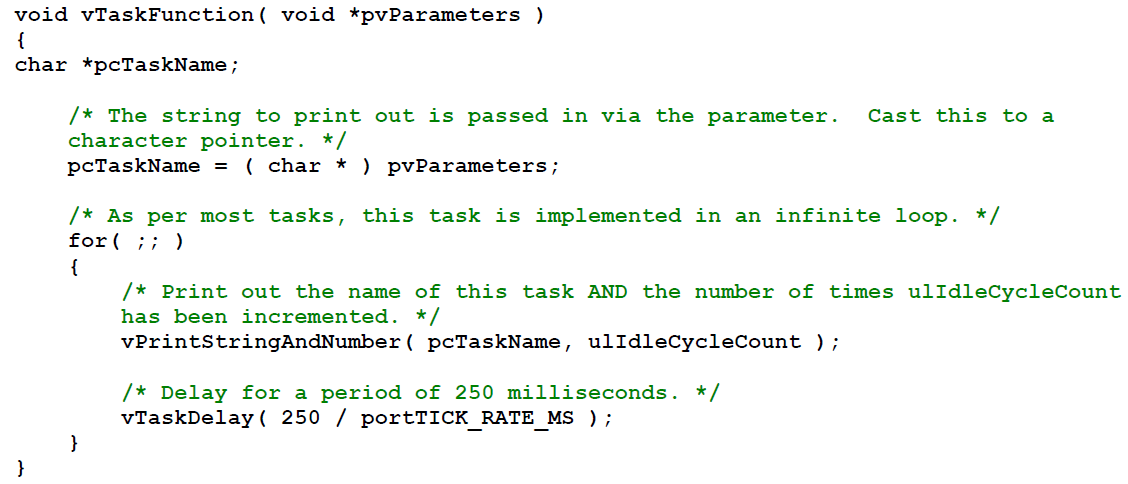
Листинг 18. Очень простой обработчик холостой задачи.



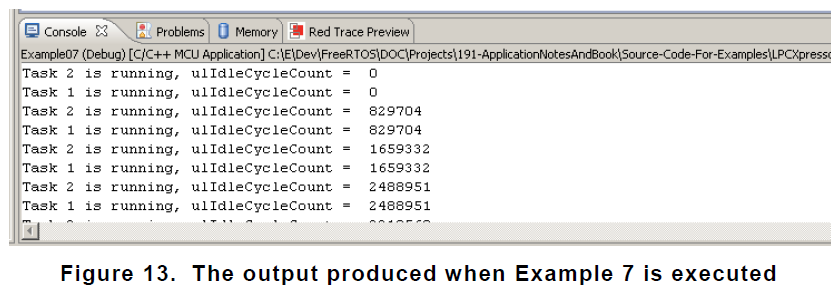
Константа конфигурации configUSE\_IDLE\_HOOK должна быть установлена в 1 внутри FreeRTOSConfig.h для того, чтобы вызывалась функция обработчика холостой задачи.

Функция, которая реализует созданные задачи немного модифицирована с целью напечатать значение ulIdleCycleCount, как показано в листинге 19.

Листинг 19. Исходный код для примера, в котором задачи выводят на печать значение ulIdleCycleCount.



Вывод, произведеный примером 7 показан на рисунке 13 и показывает, что обработчик холостой задачи вызывался примерно 830000 раз между задачами приложений.



**1.8 Изменение приоритета задачи.**

**API функция vTaskPrioritySet().**

API функция vTaskPrioritySet() может быть использована для изменения приоритета любой задачи после того, как был запущен планировщик. Заметьте, что API функция vTaskPrioritySet() доступна только, если INCLUDE\_vTaskPrioritySet установлен в 1 внутри файла FreeRTOSConfig.h.

Листинг 20. Прототип API функции vTaskPrioritySet():

void vTaskPrioritySet(xTaskHandle pxTask,

unsigned portBASE\_TYPE uxNewPriority);

Далее приведены параметры функции и возвращаемое значение.

pxTask. Дескриптор задачи, приоритет которой требуется изменить - см. параметр pxCreatedTask API функции xTaskCreate() для информации о получении дескрипторов задач. Задача может изменить ее собственный приоритет, передав NULL в качестве корректного дескриптора задачи.

uxNewPriority. Приоритет, который должен быть назначен задаче. Он ограничивается автоматически максимальным доступным уровнем приоритета (configMAX\_PRIORITIES - 1).

**API функция uxTaskPriorityGet().**

API функция uxTaskPriorityGet() может быть использована для запроса приоритета задачи. Заметьте, что эта функция доступна только когда INCLUDE\_vTaskPriorityGet установлена в 1 внутри FreeRTOSConfig.h.

Листинг 21. Прототип API функции uxTaskPriorityGet().

unsigned portBASE\_TYPE uxTaskPriorityGet(xTaskHandle pxTask);

Далее приведены параметры и возвращаемое значение функции.

pxTask. Дескриптор задачи, приоритет которой запрашивается - см. параметр pxCreatedTask API функции xTaskCreate() для информации о получении дескрипторов задач.

Задача может запросить ее собственный приоритет, передав NULL в качестве параметра корректного дескриптора задачи.

Возвращаемое значение. Текущий приоритет запрошенной задачи.

Пример 8. Изменение приоритетов задач.

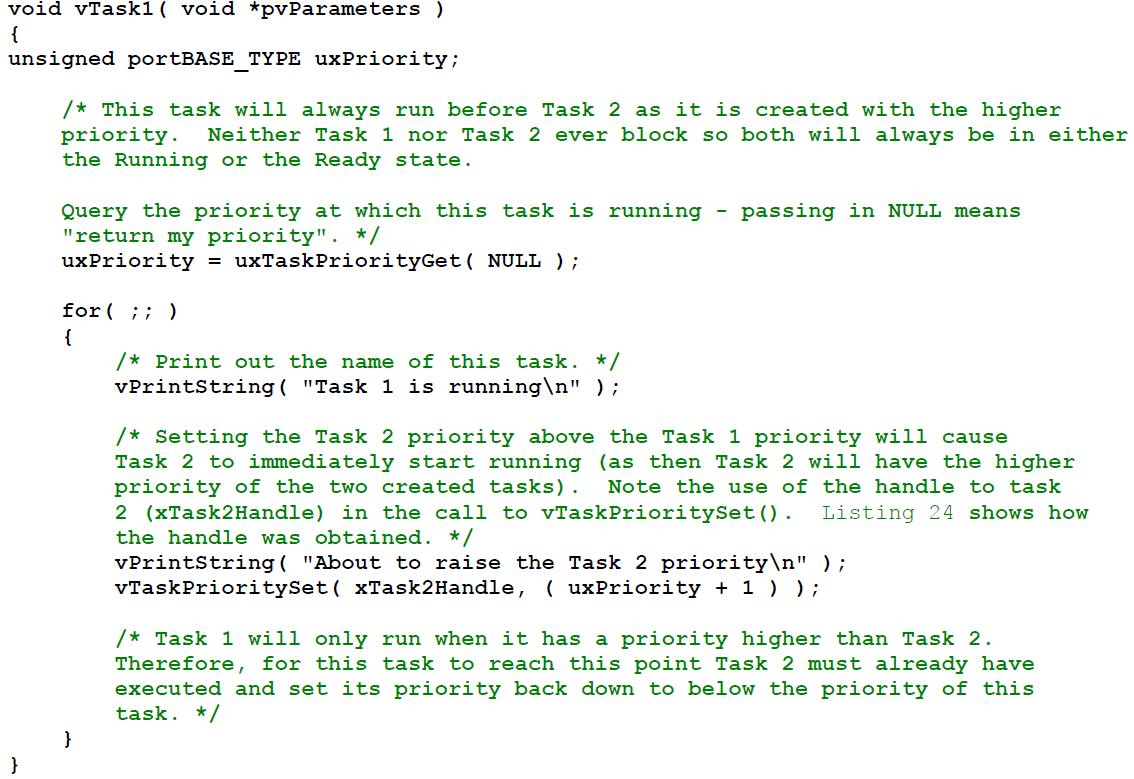
Планировщик всегда выбирает задачу с наивысшим приоритетом в состоянии "Готов" для ввода ее в состояние "Выполняется". Пример 8 демонстрирует это, используя API функцию vTaskPrioritySet() для изменения приоритетов двух задач относительно друг друга.

Создаются две задачи с одинаковыми приоритетами. Ни одна из них не использует API вызовы, которые могут заставить их войти в состояние "Блокирован", так что обе задачи всегда находятся в одном из двух состояний - либо "Готов", либо "Выполняется", следовательно, задача с наивысшим приоритетом всегда будет являться задачей, выбранной планировщиком для запуска.

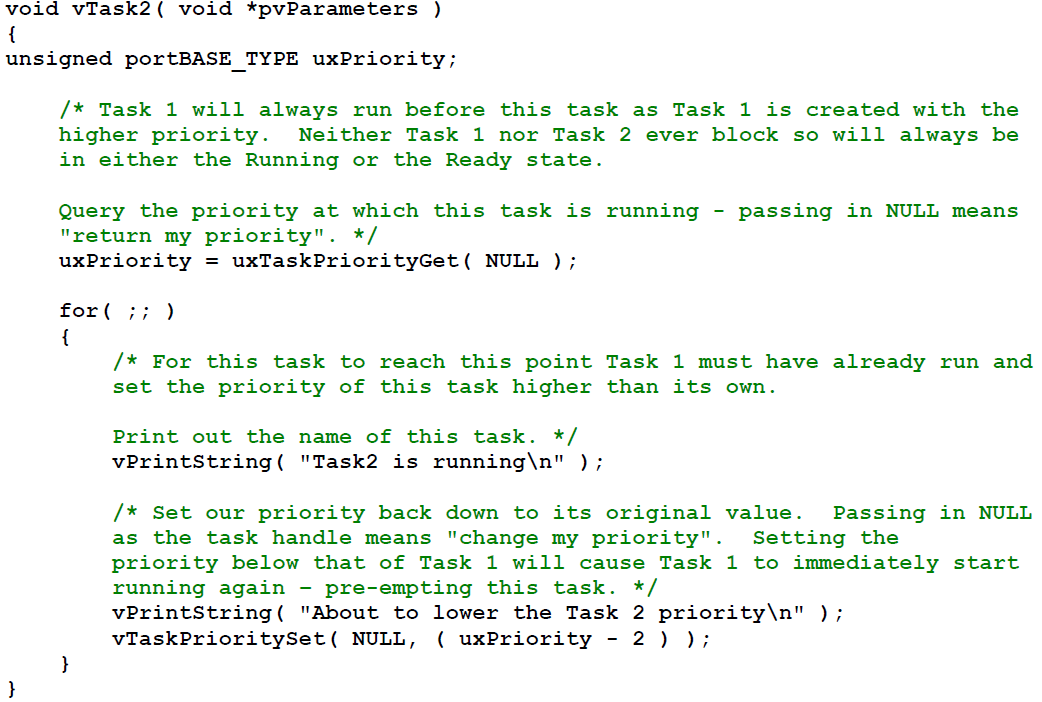
Пример 8 ведет себя следующим образом:

1. Задача 1 (листинг 22) создается с наивысшим приоритетом, что гарантирует, что она будет запущена первая. Задача 1 выводит на печать пару строк перед тем, как увеличить приоритет задачи 2 (листинг 23) выше собственного.
2. Задача 2 начинает выполняться (входит в состояние "Выполняется") сразу после того, как она получает наивысший приоритет. Только одна задача может находиться в состоянии "Выполняется" в любой момент времени, так что когда задача 2 находится в состоянии "Выполняется", задача 1 находится в состоянии "Готов".
3. Задача 2 выводит на печать сообщение перед тем, как она вернет свой приоритет обратно, ниже приоритета Задачи 1.
4. Задача 2 возвращает свой приоритет обратно перед тем, как задача 1 еще раз станет задачей с наивысшим приоритетом, так что задача 1 снова входит в состояние "Выполняется", заставляя задачу 2 вернуться обратно в состояние "Готов".

Листинг 22. Реализация задачи 1 в примере 8.



Листинг 23. Реализация задачи 2 в примере 8.



Каждая задача может как запросить, так и установить ее собственный приоритет без использования корректного дескриптора задачи, путем использования простого NULL вместо него. Дескриптор задачи требуется только когда задача ссылается на другую задачу, как например, когда задача 1 изменяет приоритет задачи 2. Чтобы позволить задаче 1 сделать это, получается и сохраняется дескриптор задачи 2, когда она создается, как выделено в комментариях листинга 24.

Листинг 24. Реализация main() в примере 8.

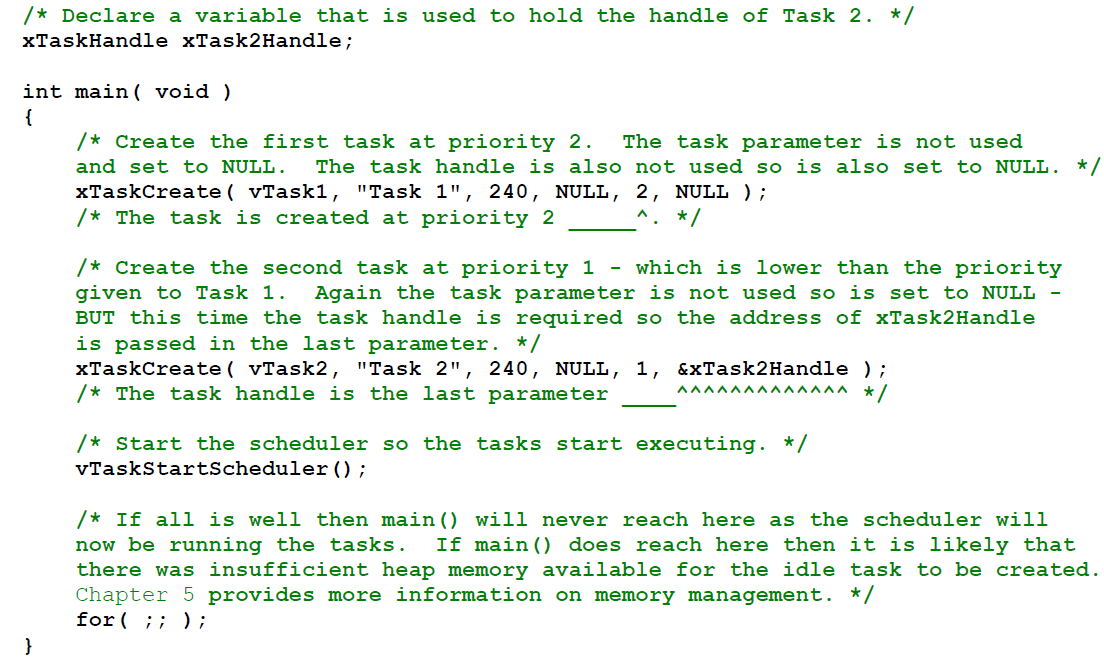
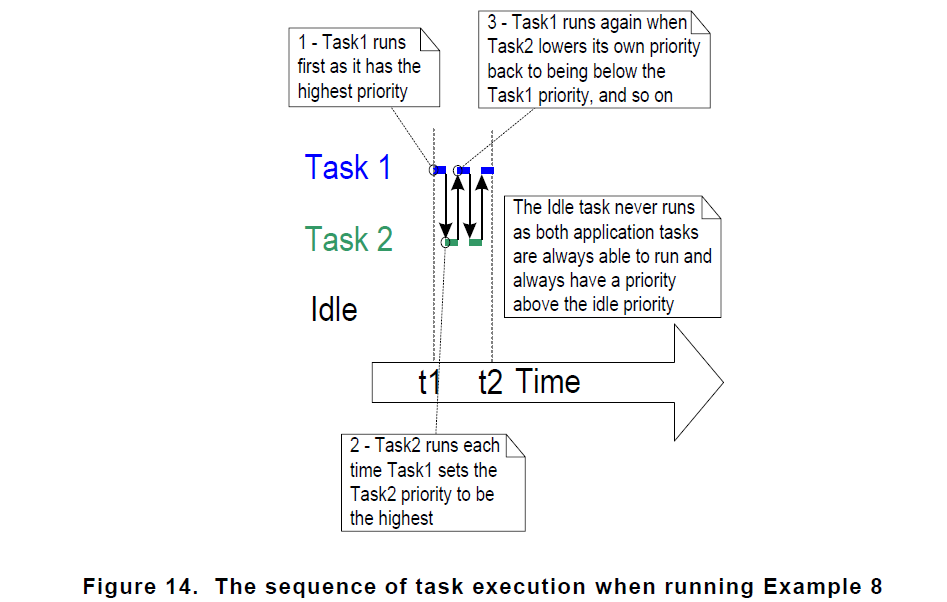
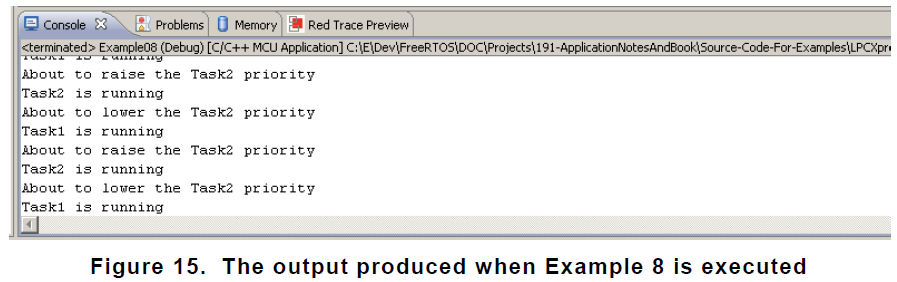


Рисунок 14 демонстрирует последовательность выполнения примера 8, результат работы примера показан на рисунке 15.





**1.9 Удаление задачи.**

**API функция vTaskDelete().**

Задача может использовать API функцию vTaskDelete() для того, чтобы удалить себя или другую задачу. Заметьте, что vTaskDelete() API функция доступна только, когда INCLUDE\_vTaskDelete установлен в 1 в файле FreeRTOSConfig.h.

Удаленные задачи больше не существуют и не могут войти снова в состояние "Выполняется". В обязанность холостой задачи входит освобождение памяти, выделенной для задач, которые были удалены. Поэтому очень важно, чтобы приложения, использующие API функцию vTaskDelete() не исчерпали ресурсы процессорного времени холостой задачи.

Заметьте, что память, выделенная задаче самим ядром будет освобождена автоматически, когда задача удаляется. Любая память или другой ресурс, выделяемые реализацией задачи самостоятельно, должны удаляться явно.

Прототип функции приведен в листинге 25.

Листинг 25. Прототип API функции vTaskDelete().

void vTaskDelete(xTaskHandle pxTaskToDelete);

Далее приведены параметры и возвращаемое значение функции.

pxTaskToDelete. Дескриптор задачи, которую требуется удалить - см. параметр pxCreatedTask API функции xTaskCreate() для информации о получении дескрипторов задач.

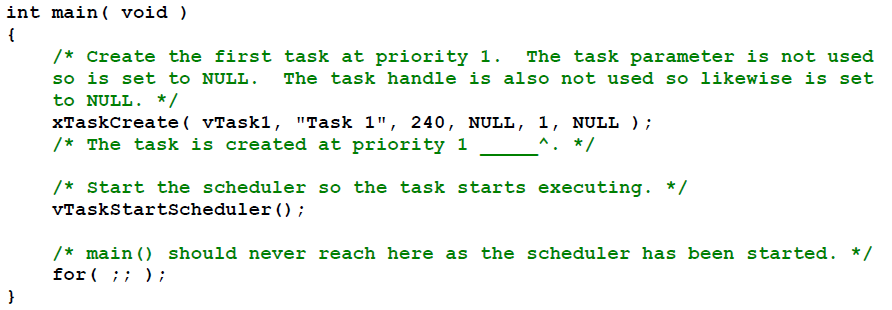
Задача может удалить саму себя, передавая NULL в качестве параметра корректного дескриптора.

Пример 9. Удаление задач.

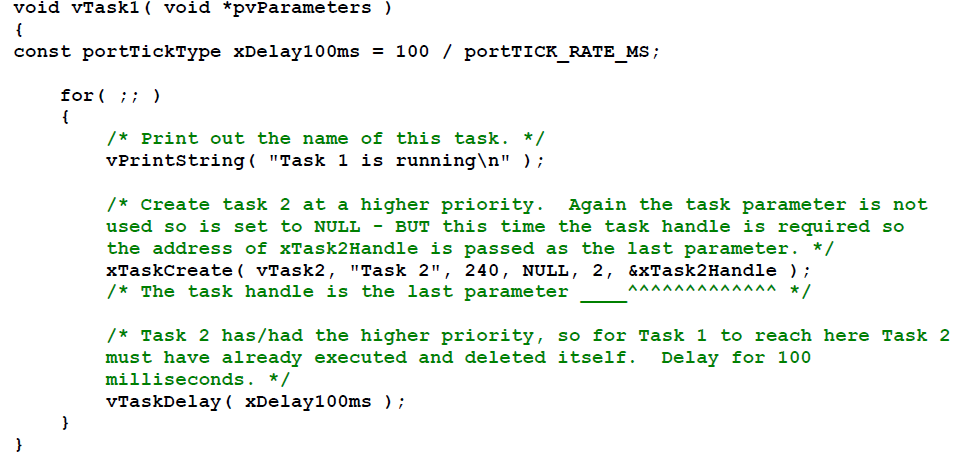
Это очень простой пример, который ведет себя следующим образом.

1. Задача 1 создается функцией main() с приоритетом 1. Когда она запускается, она создает задачу 2 с приоритетом 2. Задача 2 сейчас имеет наивысший приоритет, так что она немедленно начинает выполняться. Исходный код main() показан в листинге 26, и задача 1 в листинге 27.
2. Задача 2 ничего не делает, но удаляет себя. Она может удалить себя, передавая NULL в функцию vTaskDelete(), но вмето этого для демонстрационных целей, она использует ее собственный дескриптор. Исходный код задачи 2 показан в листинге 28.
3. Когда задача 2 была удалена, задача 1 вновь становится задачей с наивысшим приоритетом, так что продолжает выполняться - на этом этапе она вызывает vTaskDelay() для блокировки на короткий период.
4. Холостая задача выполняется пока задача 1 блокирована и освобождает память, которая была выделена под только что удаленную задачу 2.
5. Когда задача 1 покидает состояние "Блокирован", она снова становится задачей с наивысшим приоритетом в состоянии "Готов" и, таким образом, вытесняет холостую задачу. Когда она входит в состояние "Выполняется", она создает задачу 2 снова и так далее.

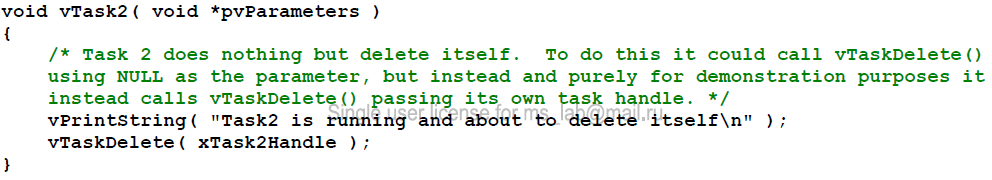
Листинг 26. Реализация функции main().

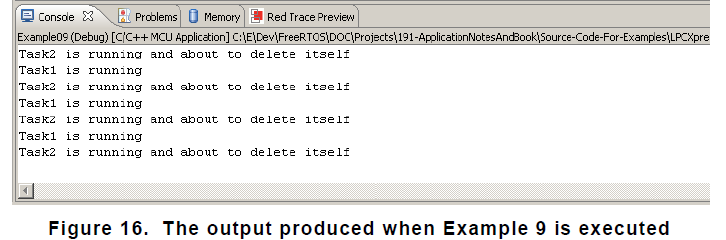


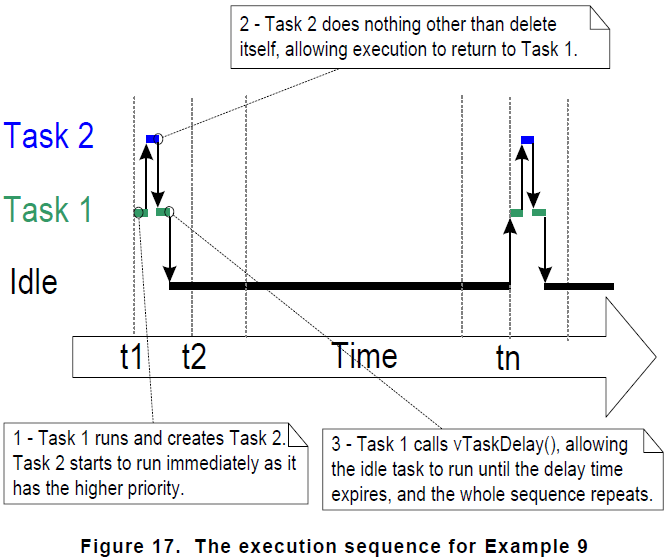
Листинг 27. Реализация задачи 1 в примере 9.



Листинг 28. Реализация задачи 2 в примере 9.







1. Задача 1 запускается и создает Задачу 2. Задача 2 начинает работать немедленно, так как имеет наивысший приоритет.
2. Задача 2 ничего не делает кроме того, что удаляет себя, позволяя задаче 1 работать дальше.
3. Задача 1 вызывает vTaskDelay(), позволяя работать холостой задаче, до тех пор, пока не истечет время задержки и вся последовательность повторится.

**1.10 Алгоритм планирования - сводка.**

**Приоритетное вытесняющее планирование.**

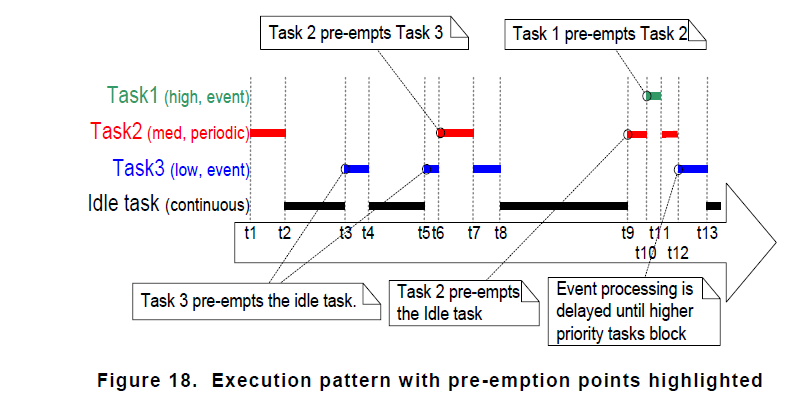
Примеры в этой главе иллюстрируют как и когда FreeRTOS выбирает какая задача должна быть в состоянии "Выполняется".

* Каждой задаче назначается приоритет.
* Каждая задача может существовать в нескольких состояниях.
* Только одна задача может существовать в состоянии "Выполняется" в любой момент времени.
* Планировщик всегда выбирает задачу с самым высоким приоритетом в состоянии "Готов", чтобы перевести ее в состояние "Выполняется".

Тип такой схемы называется "Вытесняющее планирование с фиксированным приоритетом". "Фиксированным" - потому что задаче назначается приоритет, который не изменяется самим ядром (только задачи могут изменять приоритет). "Вытесняющее" - потому, что задача, входящая в состояние "Готов", или получившая другой приоритет, всегда будет вытеснять задачу в состоянии "Выполняется", если выполняющаяся задача имеет меньший приоритет.

Задачи могут ожидать в блокированном состоянии события и автоматически возвращаться назад в состояние "Готов", когда происходит событие. Временные события происходят в определенный момент времени, например, по истечениию некоторого периода времени. В основном они используются для реализации периодического поведения или таймаутов. События синхронизации происходят, когда обработчик прерывания или задача посылают информацию в очередь, или одному из многих типов семафоров. В основном они используются для сигнализации асинхронного действия, такого как прибытие данных на периферию.

Рисунок 18 демонстрирует все это поведение, показывая шаблон выполнения гипотетического приложения.



Комментарии к рисунку 18.

1. Холостая задача. Холостая задача работает на самом низком приоритете, и вытесняется всякий раз, когда более приоритетная задача входит в состояние "Готов" - например t3, t5 и t9.
2. Задача 3. Задача 3 - это событийно-ориентированная задача, которая выполняется с относительно низким приоритетом, но выше, чем у холостой задачи. Она проводит большую часть своего времени в состоянии "Блокирован", ожидая интересуемого события и переходя в состояние "Готов" из состояния "Блокирован" всякий раз, когда происходит событие. Все механизмы межзадачного взаимодействия (очереди, семафоры и т.д.) могут быть использованы для сигнализации событий и последующей разблокировки задач. События происходят в моменты времени t3 и t5, и также где-то между t9 и t12. События, происходящие в моменты времени t3 и t5 обрабатываются немедленно, так как в эти моменты времени задача 3 является задачей с наивысшим приоритетом, которая готова к работе. Событие, которое происходит где-то между t9 и t12 не обрабатывается до момента времени t12, потому что до тех пор более высокоприоритетная задача 1 и задача 2 все еще выполняются. И только в момент t12, когда обе задачи 1 и 2 заблокированы, задача 3 становится задачей с наивысшим приоритетом, находящаяся в состоянии "Готов".
3. Задача 2. Это периодическая задача, которая выполняется с приоритетом выше, чем приоритет задачи 3, но ниже чем приоритет задачи 1. Периодический интервал подразумевает, что задача 2 хочет выполняться в моменты времени t1, t6 и t9. В момент времени t6, задача 3 находится в состоянии "Выполняется", но задача 2 имеет более высокий приоритет и вытесняет задачу 3 и начинает немедленно выполняться. Задача 2 завершает ее обработку и возвращается в состояние "Блокирован" в момент времени t7. С этого момента задача 3 может вернуться в состояние "Выполняется" чтобы завершить ее обработку. Задача 3 блокирует сама себя в момент времени t8.
4. Задача 1. Является также событийно-ориентированной задачей. Она имеет самый высокий приоритет из всех. Поэтому может вытеснить любую другую задачу в системе. Единственное показанное событие происходит в момент t10, когда Task1 вытесняет Task2. Task2 может закончить его обработку только после того, как Task1 вернется в блокированное состояние в момент времени t11.

**Выбор приоритета задачи.**

Рисунок 18 показывает фундаментальную значимость назначения приоритетов, определяющих поведение приложения.

Основное правило - задачи, которые реализуют функции с требованиями жесткого реального времени, должны иметь приоритеты выше, чем функции, реализующие задачи с требованиями мягкого реального времени. Однако, другие характеристики, такие как время выполнения и использование процессора должны также быть приняты во внимание, чтобы убедиться в том, что в пределах приложения не будет пропусков срока жесткого реального времени.

Планирование монотонного темпа (Rate Monotonic Scheduling) - это общаяя техника назначения приоритетов, которая подразумевает, что каждой задаче назначется уникальный приоритет в соответствии с темпом периодического выполнения задачи. Наивысший приоритет назначается задаче, которая имеет наивысшую частоту периодического выполнения. Самый низкий приоритет назначается задаче с самой низкой частотой периодического выполнения. Назначаемые таким образом приоритеты дают максимальную "планируемость" всего приложения, но изменения времени выполнения и тот факт, что не все задачи всегда периодические, делают абсолютные вычисления сложным процессом.

**Кооперативное планирование.**

Эта книга фокусируется на вытесняющем планировании. FreeRTOS может также опционально использовать кооперативное планирование.

Когда используется чисто кооперативный планировщик, переключение контекста будет происходить только, когда либо задача в состоянии "Выполняется" войдет в состояние "Блокирован", либо задача в состоянии "Выполняется" явно вызывает функцию taskYIELD(). Задачи никогда не будут вытесняться и задачи равных приоритетов не будут автоматически совместно использовать процессорное время. Кооперативное планирование в этом смысле проще, но, потенциально, может дать в результате менее быстро реагирующую систему.

Гибридная схема, где обработчики прерываний используются для явного принудительного переключения контекста, также имеет место быть. Это позволяет событиям синхронизации осуществлять вытеснение, но не временным событиям. В результате имеет вытесняющую систему без временных привязок. Это может понадобиться из за ее эффективности и общей конфигурации планировщика.