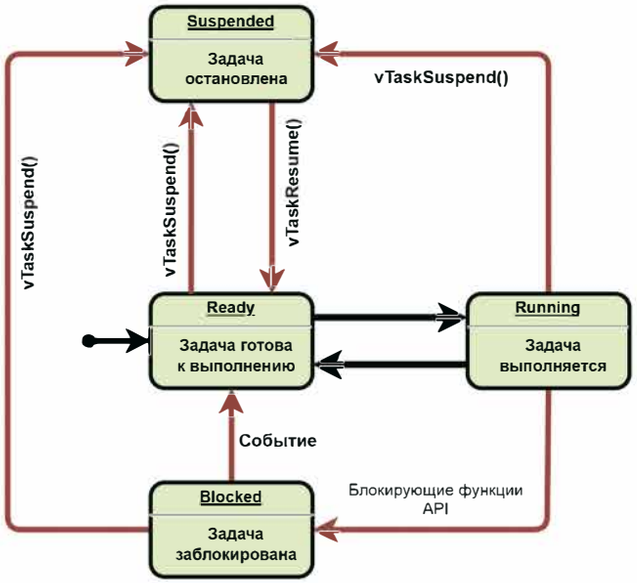
ISBN 978-5-0060-0974-5

Владимир Мединцев

ОПЕРАЦИОННЫЕ СИСТЕМЫ МИКРОКОНТРОЛЛЕРОВ

Владимир Мединцев

ОПЕРАЦИОННЫЕ СИСТЕМЫ МИКРОКОНТРОЛЛЕРОВ



ОПЕРАЦИОННЫЕ СИСТЕМЫ МИКРОКОНТРОЛЛЕРОВ

Владимир Мединцев

На примере операционной системы реального времени FreeRTOS

УДК 004
ББК 32.973
М42

Мединцев Владимир
М42 Операционные системы микроконтроллеров : На примере операционной системы реального времени FreeRTOS / Владимир Мединцев. — [б. м.] : Издательские решения, 2023. — 228 с.
ISBN 978-5-0060-0974-5

УДК 004
ББК 32.973

12+ В соответствии с ФЗ от 29.12.2010 №436-ФЗ

ISBN 978-5-0060-0974-5

Оглавление

Введение

Программирование встраиваемой электроники во многом консервативный процесс. Можно наблюдать как в мире персональных компьютеров с головокружительной скоростью происходит смена одного языка программирования другим, рождаются и уходят в бесконечность парадигмы программирования. По сравнению с этим, мир микроконтроллеров кажется островом стабильности. Отчасти это так.

Созданный в 1972 году язык Си хоть и подвергается регулярным нападкам, продолжает оставаться надежным и устойчивым стандартом в отрасли. Именно на этом языке создается более 80% кода для микроконтроллерных систем. У этого есть вполне логичное объяснение:

1. Язык Си является эффективными компилируемым языком. Создаваемые на нем программы компилируются в машинный код, что обеспечивает их высокую производительность и эффективность. Во встраиваемых системах, там, где ресурсы строго ограничены это особенно важно. Можно утверждать, что язык Си компактен. Он позволяет создавать небольшие и эффективные программы.
2. Язык Си является низкоуровневым языком. Он позволяет программистам работать на самом низком уровне абстракции, что позволяет управлять железом и оптимизировать код для каждой конкретной аппаратной платформы.
3. Си является переносимым языком, программа написанная на языке Си, может быть скомпилирована для различных архитектур и операционных систем. Это ощутимо сокращает время разработки и ресурсы необходимые для переноса проекта на новую аппаратную платформу.
4. Язык понятен разработчикам. Он понятен не только профессиональным программистам, но и инженерам-электронщикам, это сильно облегчает коммуникацию и совместную разработку на самых низких уровнях. Этим же обусловлено и наличие крупнейшего комьюнити. Наличие большого количества кода и библиотек.
5. Код написанный на языке Си легко оптимизировать.

Явные преимущества языка Си заложили основу его долголетия и популярности в отрасли. Он был и остается базовым стандартом, основой с которой начинается изучение микроконтроллеров. В связи с удобством и популярностью среди разработчиков, микроконтроллеры развивались весьма активно. Появлялись новые и совершенствовались привычные периферийные блоки. Увеличивался объём памяти, частота системной шины.

Рост количества применяемых периферийных блоков, увеличение сложности системы обработки прерываний и появление проектов большой сложности требует изменения подходов к программированию. Многим разработчикам нравится многозадачность отчасти потому, что позволяет одновременно выполнять несколько параллельных потоков без существенного усложнения кода; потому, что при использовании многозадачности становится возможным сконцентрироваться на обслуживании конкретной задачи и на написании кода. Именно такие потребности привели к тому, что в определенный момент появилось множество проектов, реализующих функции операционных систем реального времени для микроконтроллеров.

Для написания этой книги выбрана операционная система FreeRTOS.

Ядро FreeRTOS™ де-факто является стандартом на рынке. Оно разрабатывалось более 18 лет в сотрудничестве с ведущими мировыми производителями микроконтроллеров. FreeRTOS распространяется бесплатно по лицензии MIT с открытым исходным кодом и включает в себя ядро и растущий набор библиотек, подходящих для использования во всех отраслях промышленности.

FreeRTOS предлагает более низкие проектные риски и более низкую общую стоимость владения, чем коммерческие альтернативы, потому что она полностью поддерживается и документируется.

Работа над этой книгой была достаточно растянутой во времени. Какие-то главы писались во время формирования первой версии программы «Инженеров умных устройств» написанной для обучающей платформы «GeekBrains», какие-то писались позже во время пересмотра и расширения учебного курса.

Присматриваясь к идеям, заложенным в реализации функций API операционной системы, можно заметить тенденции в развитии embedded программирования за последние годы. Например, глава, посвященная

мьютексам была добавлена в последние дни работы над книгой. Мне не хотелось упоминать о мьютексах ввиду достаточно трудно отлаживаемых и диагностируемых проблем, связанных с их использованием.

Последние версии FreeRTOS показывают, что отношение разработчиков, их концептуальный взгляд на функциональное наполнение операционной системы меняется в сторону повсеместного использования нотификации и постепенного отказа от очередей, мьютексов и прочих более ресурсоемких средств синхронизации.

Тем не менее книга охватывает практически весь объём функциональных возможностей и средств рассматриваемой операционной системы.

Пользуясь возможностью, хотелось бы поблагодарить всех моих коллег, кто советами, идеями и примерами помогал создавать эту книгу. Особую признательность хотел бы выразить Александру Пономаренко, как человеку, открывшему для меня красоту FreeRTOS и подарившему идею создания этой книги и, конечно, же Эдуарду Неткачеву за поддержку проекта.

Глава 1. Суперцикл

Функция main() - основная функция создаваемого нами прикладного программного обеспечения. Именно она получает управление при запуске микроконтроллера и при нормальном функционировании создаваемого кода функция не будет завершена все время, пока работает содержащее микроконтроллер устройство. Логически функцию main() можно разделить на две части. Однократно выполняемый фрагмент кода до бесконечного цикла while и сам бесконечный цикл.

```
int main(void)
{
  // Инициализация переменных, периферии
  // Однократно выполняемый фрагмент кода
  while (1)
  {
    // Регулярно повторяющиеся действия
  }
}
```

Такая организация очень практична. В однократно выполняемую часть кода мы можем поместить вызов всех функций инициализации периферийных блоков микроконтроллера, функций инициализирующих используемые нами библиотеки и прочие действия, направленные на подготовку к выполнению основной задачи.

В качестве примера разберемся с прототипом кода, создаваемого для температурного контроллера. Функции, приведенные в примерах этой главы, имеют условные названия, передающие суть возложенного на них функционала.

```
int main(void)
{
  // Инициализация
  HAL_Init();
  SystemClock_Config();
  MX_GPIO_Init();

  while (1)
  {
          // Выполняемые программой действия
          ReadKey();      // Реакция на клавиатуру
          ReadSensor();   // Чтение значения сенсора
```

```
        DisplayData();  // Вывод данных на экран
        PID();          // ПИД регулирование
        setPWM();       // Формирование управляющего воздействия
  }
}
```

У приведенного выше примера есть один недостаток – повторение цикла происходит слишком быстро, с максимально возможной скоростью. Это не просто избыточно, иногда это может быть вредно. В большинстве случаев программист предпринимает меры к уменьшению скорости повторения цикла за счет добавления задержек.

```
  while (1)
  {
        ReadKey();
        ReadSensor();
        DisplayData();
        PID();
        setPWM();
        HAL_Delay(20);    // Задержка на 20 миллисекунд
  }
```

Добавленная задержка в 20 миллисекунд замедляет выполнение цикла и снижает частоту, с которой будет осуществляться вызов функции ReadKey() до 50 раз в секунду. Это позволяет реализовать опрос клавиатуры и борьбу с дребезгом контактов.

Однако, если мы создаем температурный контроллер, управляющий какими-либо процессами нам вряд ли потребуется получать данные о температуре и формировать управляющее воздействие с такой высокой частотой. Как следствие мы должны уменьшить частоту опроса сенсора. Теперь мы уже не можем использовать какую-либо задержку, но можем обрабатывать часть кода, пропуская несколько итераций бесконечного цикла.

```
  while (1)
  {
    ReadKey();
    if (count == 49) // Только 1 раз в 50 циклов
    {
      count = 0;
      ReadSensor();
      DisplayData();
      PID();
      setPWM();
```

```
    }
    Count++;
    HAL_Delay(20);
  }
```

После таких изменений функция ReadKey() будет вызываться 50 раз в секунду, что даст возможным функционировать алгоритму борьбы с дребезгом контактов, а все остальные функции будут вызываться 1 раз в секунду, что вполне достаточно и для осуществления процесса опроса датчика и для процесса регулирования.

Но что делать, если в этом коде нам потребуется сделать подтверждение какой-либо операции, показав пользователю пару коротких или пару длинных вспышек светодиодом? Что делать, если потребуется делать несколько операций с разной частотой вызова? К примеру, выводить информацию 1 раз в секунду, а опрашивать датчик и проводить фильтрацию считанного значения 3 раза в секунду?

В подавляющем большинстве случаев то, в чем мы действительно нуждаемся - это планировщик, некий механизм, позволяющий вызывать различные функции в заданное время или с заданной частотой.

Если посмотреть на примеры сложных проектов, размещенных на GitHub, вы найдете множество примеров того, как авторы создавали различного рода планировщики – программные реализации идеи вызова необходимых функций в необходимое время. Проще всего это можно реализовать при помощи прерывания от таймера.

Идея создания простейшего «селектора задач» - алгоритма определяющего какую функцию вызвать в каждый конкретный момент времени, может быть вполне успешно использована, но потенциально является источником ряда практически не решаемых проблем:

- Если какая-либо функция получающая управление от простейшего «селектора задач» требует продолжительного времени для выполнения, все остальные действия будут отложены;
- Нет возможности менять выполняемую функцию (выполняемую задачу) в режиме реального времени в ответ на внешние события;
- Нет возможности обеспечить выделение пропорционального времени каждой вызываемой функции.

Иными словами, если какой-либо функции вызываемой таким импровизированным «селектором задач» потребуется дополнительно время в результате возникшего «таймаута», то это время будет предоставлено в ущерб другим функциям. Это является главной проблемой такого рода подхода.

FreeRTOS™

Операционная система реального времени (ОСРВ, англ. Real-time operating system, RTOS) — тип специализированной операционной системы, основное назначение которой — предоставление необходимого и достаточного набора функций для проектирования, разработки и функционирования систем реального времени на конкретном аппаратном оборудовании.

FreeRTOS – один из примеров реализации операционных систем реального времени для приложений, использующих микроконтроллеры. Первоначально FreeRTOS разрабатывалась компанией Real Time Engineers Ltd.

Приложения, создаваемые для микроконтроллеров чаще всего, включают в себя сочетания как жестких, так и мягких требований реального времени.

Требования мягкого реального времени — это те, которые устанавливают крайний срок реакции на возникающие события, но нарушение крайнего срока не делает систему бесполезной. К примеру, слишком медленная реакция на нажатие клавиши может привести к тому, что система будет казаться немного заторможенной, но эта особенность не сделает ее непригодной для использования. Да систем мягкого реального времени характерно следующее:

- Система имеет гарантированное время реакции на внешние события (прерывания от оборудования);
- жёсткая подсистема планирования процессов (высокоприоритетные задачи не должны вытесняться низкоприоритетными, за некоторыми исключениями);
- повышенные требования к времени реакции на внешние события или реактивности (задержка вызова обработчика

прерывания не более десятков микросекунд, задержка при переключении задач не более сотен микросекунд).

Жесткие требования реального времени — это те, которые устанавливают крайний срок реакции на событие, и нарушение этого срока приведет к полному отказу системы. Например, подушка безопасности водителя может принести больше вреда, чем пользы, если она слишком медленно реагирует на сигналы от датчика столкновения.

Спецификация UNIX в редакции 2 даёт следующее определение: Реальное время в операционных системах — это способность операционной системы обеспечить требуемый уровень сервиса в определённый промежуток времени.

Основа FreeRTOS — это ядро реального времени (или планировщик реального времени), на основе которого можно создавать встроенные приложения, отвечающие требованиям жесткого реального времени. Это позволяет организовать приложения как набор независимых потоков выполнения. На процессоре с одним ядром одновременно может выполняться только один поток. Ядро решает, какой поток должен выполняться, изучая приоритет, назначенный каждому потоку разработчиком приложения. В простейшем случае разработчик приложения может назначить более высокий приоритет потокам, реализующим требования жесткого реального времени, и более низкий приоритет потокам, реализующим требования мягкого реального времени. Это гарантирует, что потоки жесткого реального времени всегда будут выполняться раньше потоков мягкого реального времени, однако и решения о назначении приоритетов не всегда так просты.

В начале этой главы, в качестве примера, иллюстрирующего необходимость создания некоторого диспетчера обеспечивающего выполнения определенных программных функций в заданное время, приведен пример реализующий идеи приоритизации задач. В реальных проектах использование ядра ОСРВ может принести и другие, менее очевидные преимущества:

- Абстрагирование от времени. Ядро отвечает за синхронизацию выполнения и предоставляет приложению, связанный со временем API. Это позволяет упростить структуру кода приложения и уменьшить общий размер кода.

- Расширяемость. Абстрагируя код от деталей синхронизации, мы порождаем меньшее количество взаимозависимостей между модулями. Это дает возможность контролируемого развития ПО.
- Модульность. Задачи — это самостоятельные модули, каждый из которых должен иметь четко определенную цель.
- Командная работа. Разделение кода на задачи облегчает разработку приложения в команде.
- Упрощение тестирования. Если задачи представляют собой четко определенные независимые модули с понятными интерфейсами, их можно тестировать изолированно.
- Реиспользование кода. Благодаря большей модульности и меньшему количеству взаимозависимостей код можно повторно использовать.
- Повышение эффективности. За счет использования ядра процессорное время не используется впустую. Задачи, которые не должны выполняться не будут занимать процессорное время.
- Утилизация времени. Задача Idle создается автоматически при запуске планировщика. Она выполняется всякий раз, когда появляется свободное (не востребованное другими задачами) процессорное время. Idle можно использовать для измерения свободных вычислительных мощностей, выполнения фоновых проверок или просто для перевода процессора в режим пониженного энергопотребления.
- Управление питанием. Повышение эффективности, достигаемое при использовании RTOS, позволяет процессору проводить больше времени в режиме пониженного энергопотребления.
- Отложенная обработка прерывания. Обработчики прерываний можно сделать очень короткими, перенеся обработку либо в задачу, созданную автором приложения, либо в задачу демона FreeRTOS.

Самый простой пример, иллюстрирующий подход, предлагаемый системой реального времени, это приложение, которое включает клавиатуру и ЖК-дисплей. Пользователь должен получать визуальную обратную связь о каждом нажатии клавиши в течение разумного периода времени - если пользователь не видит, что нажатие клавиши было принято в течение этого периода, программным продуктом будет в лучшем случае неудобно пользоваться. Если наиболее длительный

допустимый период составлял 100 мс - приемлемым будет любой ответ от 0 до 100 мс. Эта функция может быть реализована как автономная задача со следующей структурой:

```
void vKeyHandlerTask( void *pvParameters )
 {
     // Обработка нажатий на клавиши это непрерывный процесс
     // по этой причине задача представляет собой бесконечный
     // цикл. Как и большинство задач в реальном времени.
     for( ;; )
     {
         [Ожидание нажатия на клавишу]
         [Обработка нажатой клавиши]
     }
 }
```

Теперь предположим, что система реального времени также выполняет функцию управления, основанную на входных данных с цифровой фильтрацией. Входной сигнал должен быть дискретизирован, отфильтрован, а цикл управления должен выполняться каждые 2 мс. Для правильной работы фильтра временная регулярность выборки должна быть с точностью до 0,5 мс. Эта функция может быть реализована как автономная задача со следующей структурой:

```
void vControlTask( void *pvParameters )
 {
     for( ;; )
     {
         [Пауза на 2ms с момента прошлого цикла]

         [Получение входных данных]
         [Фильтрация входных данных]
         [Алгоритм управления]
         [Выходное воздействие]
     }
 }
```

Программист должен назначить задаче управления наивысший приоритет:

1. Частота выполнения и требовательность ко времени у задачи управления более жесткий, чем у задачи контроля клавиатуры.

2. Последствия временной задержки связанной с задачей управления больше, нежели чем у задачи связанной с обработкой нажатий на клавиши.

Терминология

Задача - Во FreeRTOS каждый поток выполнения называется «задачей». В отличии от сообщества Linux/Unix систем в embedded нет единого мнения относительно терминологии. Это будет особенно заметно если после прочтения официальной документации FreeRTOS вы прочитаете документацию Microsoft на ThreadX. Однако, лично я предпочитаю «задачу» «потоку», поскольку в некоторых областях применения поток может иметь более конкретное значение.

API (Application Programming Interface) - интерфейс программирования приложений. Это совокупность инструментов и функций, которые представляют собой интерфейс для создания новых приложений. И это замечательное определение из мира ПК. В нашем случае мы имеем дело с кодом написанным на языке Си для микроконтроллеров. И операционная система, и драйвера периферийных устройств (HAL), и создаваемое прикладное программное обеспечение будут выполняться в едином адресном пространстве микроконтроллера. По этой причине понятие API сводится к набору функций, определенных в коде операционной системе и предназначенных для взаимодействия ОС, и прикладного программного обеспечения.

RTOS - Операционная система реального времени (ОСРВ, англ. real-time operating system) — тип специализированной операционной системы, основное назначение которой — предоставление необходимого и достаточного набора функций для проектирования, разработки и функционирования систем реального времени на конкретном аппаратном оборудовании.

Портирование – адаптация некоторой программы или её части, чтобы она работала в другой среде, отличающейся от той среды, под которую она была изначально написана с максимальным сохранением её пользовательских свойств. Процесс портирования также называют портированием или переносом, а результат — портом. Но в любом случае главной задачей при портировании является сохранение привычных

пользователю интерфейса и приёмов работы с программой. Добавление новых или удаление части имеющихся свойств при портировании программных продуктов не допускается. Необходимость в портировании возникает обычно из-за различий в системе команд процессора, различий между способами взаимодействия операционной системы и программ, принципиальных различий в архитектуре вычислительных систем, либо по причине некоторых несовместимостей или даже полного отсутствия используемого языка программирования в целевом окружении.

Глава 2. Структура FreeRTOS

Ядро FreeRTOS и другие библиотеки FreeRTOS распространяются бесплатно по лицензии MIT с открытым исходным кодом (SPDX-License-Identifier: MIT). Это означает, что вы можете использовать FreeRTOS:

- Абсолютно бесплатно;
- Даже в коммерческих приложениях – бесплатно;
- Никакая гарантия не предоставляется;
- Использование FreeRTOS не накладывает на вас обязательств по открытию исходного кода своего приложения.
- Вы не обязаны открывать исходный код своих изменений, вносимых в ядро FreeRTOS;
- Вы не обязаны каким-либо образом документально подтверждать, что ваш продукт использует FreeRTOS;
- Не обязаны предлагать или предоставлять код FreeRTOS пользователям вашего приложения.

Исходный код FreeRTOS доступен на сайте: https://www.freertos.org

У лицензии MIT есть интересная особенность. Несмотря на тот факт, что лицензия содержит слова «безвозмездно использовать Программное Обеспечение без ограничений» для получения разрешения необходимо выполнить фразу «при соблюдении следующих условий» … «Указанное выше уведомление об авторском праве и данные условия должны быть включены во все копии или значимые части данного Программного Обеспечения.» По сути это означает, что всем пользователям, которым вы предоставите возможность использовать разрабатываемое вами программное обеспечение вы должны сообщить, что его часть свободно распространяется по лицензии MIT, и что эта часть называется FreeRTOS.

Помимо FreeRTOS на сайте доступны еще две версии операционной системы, реализованные на ее основе:

1. OpenRTOS — это версия ядра FreeRTOS с коммерческой лицензией, которая включает возмещение убытков и специальную поддержку. FreeRTOS и OpenRTOS используют одну и ту же кодовую базу.
2. SafeRTOS — это производная версия ядра FreeRTOS, которая была проанализирована, задокументирована и протестирована

на соответствие строгим требованиям промышленного (IEC 61508 SIL 3), медицинского (IEC 62304 и FDA 510(K)), автомобильного (ISO 26262) и других международных стандарты безопасности. SafeRTOS включает артефакты документации по жизненному циклу безопасности, прошедшие независимую проверку.

FreeRTOS спроектирована так, чтобы быть простой и удобной в использовании: требуются только 3 исходных файла, которые являются общими для всех портов RTOS, и один исходный файл для конкретного микроконтроллера. API файла порта разработан таким образом, чтобы быть простым и интуитивно понятным. Каждый официальный порт поставляется с официальным демонстрационным примером, который (по заверению разработчиков) компилируется и выполняется на аппаратной платформе, для которой данный пример предназначен, без каких-либо изменений.

Если рассматривать развертывание операционной системы FreeRTOS применительно к микроконтроллерам STM32, то вся работа по подключению необходимых файлов к проекту, изменению конфигурационного файла FreeRTOSConfig.h и созданию основы проекта будет возложена на конфигуратор кода STM32CubeMX.

НЕОБХОДИМО ОСОБО ОТМЕТИТЬ ВАЖНУЮ ДЕТАЛЬ – При использовании генератора кода STM32CubeMX или интегрированной среды разработки STM32CubeIDE для создания проекта помимо очевидного ядра операционной системы FreeRTOS Kernel Copyright (C) 2020 Amazon.com в проект будут добавлены и файлы проекта CMSIS-RTOS API. Данные файлы представляют собой ничто иное, как порт выполненный компанией STMicroelectronics.

В результате расширенного портирования будут доступны как официально предоставляемые функции API операционной системы, так и расширенные функции, написанные с учетом особенностей микроконтроллеров, производимых STMicroelectronics. Более подробно с информацией об особенностях применения CMSIS-RTOS API можно ознакомиться, прочитав UM1722 - Developing applications on STM32Cube with RTOS доступный на сайте STMicroelectronics.

Если вы решите использовать STM32CubeMX в качестве генератора кода, то внедрение файлов операционной системы в создаваемый проект

будет осуществлено без каких-либо сложностей. Нужно выполнить несколько простых действий:

1. Выберите категорию Middleware и щелкните по библиотеке FREERTOS.
2. Выберите интерфейс CMSIS версия 1 или версия 2.

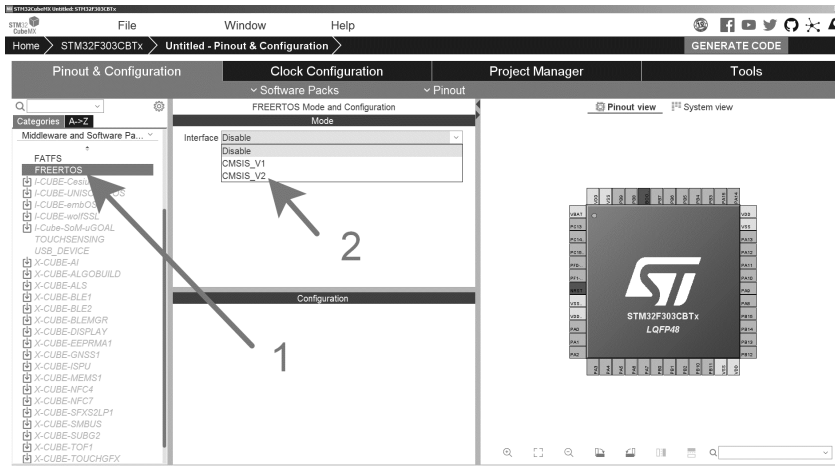


Рисунок 1. Интерфейс программы STM32CubeMX. Подключение FreeRTOS.

3. Вам станут доступны вкладки конфигурации параметров операционной системы, настройки задач, очередей, таймеров и прочих элементов.

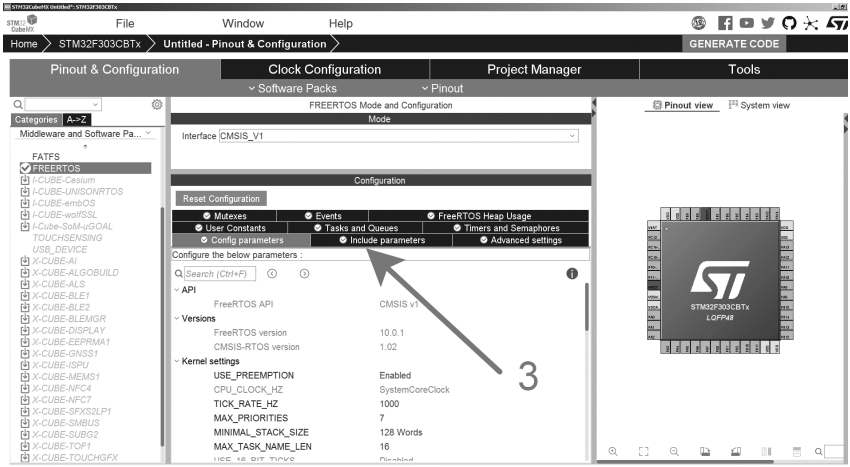


Рисунок 2. Интерфейс программы STM32CubeMX. Подключение FreeRTOS.

Это вовсе не означает, что процесс самостоятельной интеграции операционной системы в уже существующий проект без использования генератора кода, является затратным по времени или сложным процессом.

Прежде всего потребуется скачать архив с файлами операционной системы с официального сайта проекта.

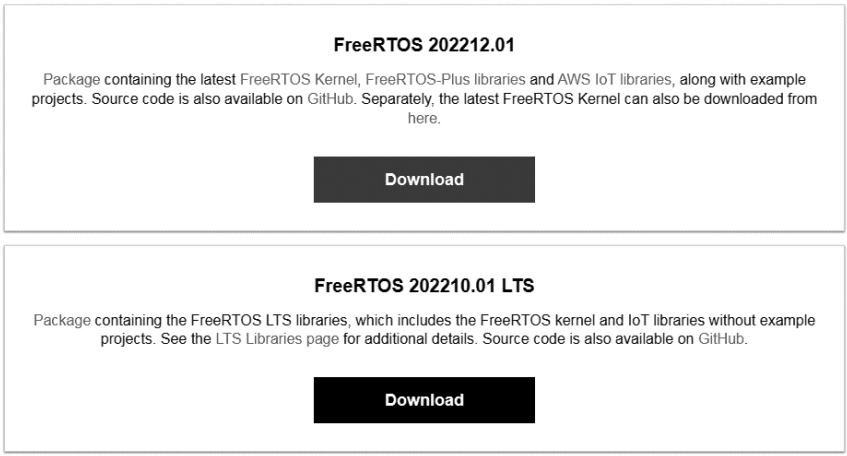


Рисунок 3. Раздел «Загрузка» сайта freertos.org

Файлы FreeRTOS

Структура операционной системы позволяет с легкостью портировать ее для использования примерно с двадцатью различными компиляторами и может работать на более чем тридцати различных процессорных архитектурах. Каждая поддерживаемая комбинация компилятора и процессора считается отдельным портом FreeRTOS.

Упрощенно, FreeRTOS можно рассматривать как библиотеку, предоставляющую возможности многозадачности. FreeRTOS поставляется в виде набора исходных файлов на языке Си. Некоторые из исходных файлов являются общими для всех портов, в то время как другие специфичны для порта.

Интеграция FreeRTOS в проект заключается во включении исходных файлов операционной системы как часть вашего проекта, тем самым вы делаете API операционной системы доступным для вашего приложения.

FreeRTOS настраивается с помощью конфигурационного файла FreeRTOSConfig.h. Этот конфигурационный файл содержит в себе константы, макросы и определения необходимые для гибкого конфигурирования операционной системы под нужды конкретного приложения.

Обратите внимание что для каждого порта операционной системы написано собственное демонстрационное приложение, которое уже содержит пример конфигурационного файла FreeRTOSConfig.h. По этой причине нет необходимости создавать файл FreeRTOSConfig.h с нуля. Вместо этого рекомендуется начать с FreeRTOSConfig.h, используемого демонстрационным приложением, предоставленным для используемого порта FreeRTOS, а затем адаптировать его.

Официально FreeRTOS распространяется в одном zip-файле. ZIP-файл содержит исходный код для всех портов FreeRTOS и файлы проектов для всех демонстрационных приложений FreeRTOS. Он также содержит набор компонентов экосистемы FreeRTOS+ и набор демонстрационных приложений экосистемы FreeRTOS+. Пусть вас не смущает количество файлов в дистрибутиве FreeRTOS! В любом приложении требуется очень небольшое количество файлов.

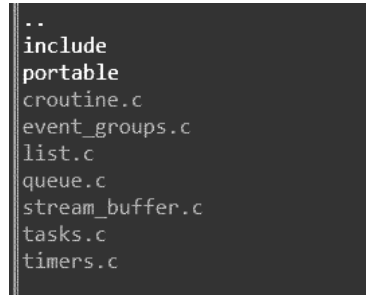


Рисунок 4. Список файлов.

На рисунке 4 показаны файлы операционной системы. Файлы расположены в каталоге FreeRTOSv202112.00\FreeRTOS\Source.

Давайте начнем знакомится с файлами, общими для всех существующих портов операционной системы. Основной исходный код FreeRTOS содержится всего в двух файлах Си. Они называются tasks.c и list.c и расположены непосредственно в каталоге FreeRTOS/Source. Помимо этих двух файлов, в том же каталоге находятся следующие исходные файлы:

- queue.c – файл queue.c предоставляет службы очередей и семафоров и активно используется почти во всех проектах.
- timers.c – предоставляет функциональные возможности для использования программных таймеров. Его нужно включать в сборку только в том случае, если программные таймеры действительно будут использоваться.
- event_groups.c - предоставляет функциональные возможности группы событий.
- croutine.c - реализует функциональность сопрограммы. Изначально сопрограммы задумывались для работы на микроконтроллерах с крайне ограниченными ресурсами, в настоящее время применяются редко.
- stream_buffer.c – это новый функционал в операционной системе. Потоковые буферы используются для отправки непрерывного потока данных от одной задачи или прерывания к другой. Их реализация достаточно компактна, что делает их особенно подходящими для сценариев отправки данных из обработчика прерывания в задачу и (или) между ядрами.

Помимо описанных выше файлов вам обязательно потребуется файл порта, соответствующий используемому вами компилятору и архитектуре микроконтроллера. Эти файлы содержатся в каталоге FreeRTOS/Source/portable. Искомые файлы порта будут находиться в каталоге FreeRTOS/Source/portable/[compiler]/[architecture].

Как правило в качестве порта, в каталоге будут находиться файлы port.c и portmacro.c. Эти файлы содержат определения (define) и макросы, позволяющие операционной системе низкоуровнево взаимодействовать с микроконтроллером. Это и взаимодействие с системным таймером SysTick и определения типов, используемых данных, как элемент переносимости кода под особенности компилятора.

Очень важно заметить, что и взаимодействие между операционной системой и памятью микроконтроллера также является частью переносимого уровня. В следующей главе мы обязательно разберемся с тем, как происходит выделение памяти под задачи.

На текущий момент FreeRTOS предоставляет пять примеров схем распределения кучи. Пять схем называются от heap_1 до heap_5 и реализуются исходными файлами от heap_1.c до heap_5.c соответственно. Примеры схем распределения кучи содержатся в каталоге FreeRTOS/Source/portable/MemMang. Если вы настроили FreeRTOS для использования динамического выделения памяти, то необходимо скопировать один из данных файлов включив их в состав создаваемого вами проекта.

Необходимо особо отметить, что примеры распределения кучи имеют смысл только в тех приложениях, которые содержат константу configSUPPORT_DYNAMIC_ALLOCATION, имеющую значение 1. Константа должна быть определена в конфигурационном файле FreeRTOSConfig.h.

Как следует из изложенного выше, для интеграции FreeRTOS в создаваемый вами проект потребуется включить всего три каталога в пути компилятора. Это:

1. Путь к основным заголовочным файлам - FreeRTOS/Source/include
2. Путь к файлам порта - FreeRTOS/Source/portable/[compiler]/[architecture]
3. Путь к конфигурационному файлу проекта - FreeRTOSConfig.h.

После того, как пути к обозначенным файлам прописаны и могут быть обработаны компилятором в файл с исходным кодом вашего проекта необходимо добавить следующие include:

```
#include "FreeRTOS.h"
#include "task.h"
#include "timers.h"
#include "queue.h"
#include "semphr.h"
#include "event_groups.h"
```

Первая строка - #include "FreeRTOS.h" добавляется в любом случае. Все прочие файлы заголовков добавляются только в том случае, если вы используете их функционал в своем проекте.

Обратите внимание, если вы используете микроконтроллеры STMicroelectronics и генератор кода ST32CubeMX для создания основы проекта, то вместо #include "FreeRTOS.h" генератор добавит в ваш проект только включение заголовочного файла:

```
#include "cmsis_os.h"
```

Если обратиться к документации разработчиков на операционную систему FreeRTOS, то они рекомендуют начинать создание нового проекта с адаптации распространяемых вместе с ОС примеров. Это вполне сносный совет, когда вы начинаете освоение новой технологии. Если же чтение сообщений об ошибках компилятора не представляет для вас большой проблемы, то можно написать файл main.c проекта и самостоятельно.

1. Используя выбранные вами инструменты, создайте новый проект, который еще не включает в себя исходные файлы FreeRTOS.
2. Убедитесь в том, что проект собирается, загружается на целевой микроконтроллер и выполняется.
3. Добавьте исходные файлы *.c в созданный вами проект. Как было отмечено выше вам обязательно потребуются файлы tasks.c и list.c. Остальные файлы необходимо добавить в проект при необходимости использования их функционала.

4. Скопируйте из демонстрационного примера порта конфигурационный файл FreeRTOSConfig.h и разместите его в своем проекте.
5. Добавьте в пути вашего проекта маршрут к следующим папкам, содержащим файлы заголовков:
    a. FreeRTOS/Source/include
    b. FreeRTOS/Source/portable/[compiler]/[architecture]
    c. Каталог содержащий конфигурационный файл FreeRTOSConfig.h
6. Сконфигурируйте все необходимые обработчики прерываний FreeRTOS. Используйте веб-страницу описывающую используемый вами порт и демонстрационный проект как образец.

В качестве примера, взгляните на код обработчика прерывания системного таймера SysTick. Как видите, если планировщик запущен, будет вызвана функция xPortSysTickHandler().

```
void SysTick_Handler(void)
{
  HAL_IncTick(); /* Увеличение счетчика в библиотеке HAL 0 */
  if (xTaskGetSchedulerState() != taskSCHEDULER_NOT_STARTED)
  {
     xPortSysTickHandler();
  }
}
```

Код ниже демонстрирует возможный код функции main() с запуском планировщика.

```
int main( void )
{
  /* Конфигурируем и инициализируем периферию. */
  mySetupHardware();
  /* Создаем задачи, очереди и пр. обьекты ОСРВ */
  /* Запускаем планировщик */
  vTaskStartScheduler();
  /* Мы достигнем этой части кода только в том случае,
     если планировщик аварийно завершит свою работу. */
  for( ;; );
  return 0;
}
```

Типы данных и стиль

Гораздо легче понимать написанное сторонними разработчиками ПО или библиотеку, если знать используемый стиль именования функций, переменных и предпочитаемые типы данных. Для операционной системы FreeRTOS существует набор крайне простых и понятных правил.

Каждый вариант адаптации (port) FreeRTOS под конкретный микроконтроллер имеет уникальный заголовочный файл portmacro.h, который содержит (помимо прочего) определения для двух специальных типов данных, TickType_t и BaseType_t.

TickType_t - Используется для хранения значения счетчика тиков и переменных, которые задают время блока. TickType_t может быть 16-битным без знака или 32-битным без знака, в зависимости от настройки configUSE_16_BIT_TICKS в конфигурационном файле FreeRTOSConfig.h.

Использование 16-битного типа может значительно повысить эффективность в 8-битных и 16-битных архитектурах, но сильно ограничивает максимальный период таймаута, который может быть задан. Нет причин использовать 16-битный тип данных в 32-битной архитектуре микроконтроллеров.

BaseType_t - Всегда определяется как наиболее эффективный тип данных для архитектуры. Обычно это 32-битный тип в 32-битной архитектуре, 16-битный тип в 16-битной архитектуре и 8-битный тип в 8-битной архитектуре. BaseType_t обычно используется для переменных, которые могут принимать только очень ограниченный диапазон значений, а также для логических значений. Стандартные типы данных, отличные от char, не используются, вместо этого используются имена типов, определенные в файле заголовка stdint.h компилятора. Типы «char» могут указывать только на строки ASCII или ссылаться на отдельные символы ASCII.

Имена переменных

В качестве первых символов имени переменной используются префиксы, отражающие тип данной переменной: «c» для char, «s» для short, «l» для long и «x» для BaseType_t и любых других типов (структуры, дескрипторы задач, дескрипторы очереди и т. д.).

Если переменная беззнаковая, она также имеет префикс «u». Если переменная является указателем, она также имеет префикс «p». Следовательно, переменная типа unsigned char будет иметь префикс «uc», а переменная типа указатель на char будет иметь префикс «pc».

Имена функций

Имена функций имеют префикс, отражающий как тип данных, который они возвращают, так и префикс, соответствующий имени файла, в котором они определены. Например:

- vTaskPrioritySet () возвращает void и функция определена в файле task.c.
- xQueueReceive () возвращает значение типа BaseType_t и определена в файле queue.c.
- pvTimerGetTimerID() возвращает указатель (pointer) на void и определена в файле timers.c.

Функции, используемые исключительно внутри файла, имеют префикс «prv».

Форматирование

Один отступ табуляции всегда равен 4-м пробелам.

Макросы

Имена большинства макросов пишутся в верхнем регистре и имеют префикс, набранный строчными буквами. Префикс определяет файл, в котором дано определение макроса. Рассмотрим примеры префиксов используемых в именовании макросов:

- port (например, portMAX_DELAY) макрос определен в файлах portable.h или portmacro.h
- task (например, taskENTER_CRITICAL()) определен в файле task.h
- pd (например, pdTRUE) определен в файле projdefs.h
- config (например, configUSE_PREEMPTION) определен в конфигурационном файле FreeRTOSConfig.h
- err (например, errQUEUE_FULL) определен в файле projdefs.h

У описанного есть всего одно исключение. Практически весь API работы с семафорами написан с использованием макросов, однако

имена макросов семафоров используют правила, описанные выше для функций.

Также существуют глобальные определения доступные во всем исходном коде FreeRTOS это определения: pdTRUE = 1, pdFALSE = 0, pdPASS = 1 и pdFAIL = 0.

Глава 3. Управление памятью

Начиная с FreeRTOS V9.0.0, мы можем использовать две модели размещения объектов ядра в памяти: Статически и Динамически. Статически, когда вся необходимая для размещения объекта память выделяется ему статически во время компиляции. Динамически – память выделяется во время выполнения.

Во всех последующих главах этой книги мы будем изучать такие объекты ядра, как задачи, очереди, семафоры и группы событий. Чтобы максимально упростить использование FreeRTOS, эти объекты ядра не размещаются статически во время компиляции, а динамически распределяются во время выполнения; FreeRTOS выделяет ОЗУ каждый раз, когда создается объект ядра, и освобождает ОЗУ каждый раз, когда объект ядра удаляется. Эта политика сокращает усилия по проектированию и планированию, упрощает API.

В этой главе, мы будем разбираться именно с особенностью реализации динамического выделения памяти. Необходимо заметить, что вопрос выделения памяти это всего лишь концепция программирования на Си. Этот вопрос не является чем-то специфичным именно для FreeRTOS или многозадачности. Однако, мы говорим об этом в концепте операционной системы FreeRTOS, поскольку объекты ядра выделяются динамически, а схемы динамического выделения памяти, предоставляемые компиляторами общего назначения, не всегда подходят для приложений реального времени.

Чаще всего, когда возникает дискуссия о выделении и последующем высвобождении памяти, мы вспоминаем что эти операции могут быть легко проделаны с помощью стандартных функций malloc() и free() библиотеки Си. В зависимости от ядра микроконтроллера и версии компилятора эти функции могут стать не самым лучшим решением по одной или даже нескольким причинам:

- Они не всегда доступны в небольших встроенных системах.
- Их программная реализация может быть относительно большой, занимая ценное место в коде.
- Они редко потокобезопасны.

- Они, как правило, не детерминированы; количество времени, необходимое для выполнения функций, будет отличаться от вызова к вызову.
- Они могут страдать от фрагментации.
- Они могут усложнить настройку компоновщика.
- Они могут быть источником трудно отлаживаемых ошибок, если пространство кучи может увеличиваться в памяти, используемой другими переменными.

Как отмечалось выше, начиная с FreeRTOS V9.0.0, объекты ядра могут размещаться статически во время компиляции или динамически во время выполнения. Теперь FreeRTOS рассматривает выделение памяти как часть переносимого уровня, как часть порта. Это связано с тем фактом, что разные встраиваемые системы имеют разные требования к распределению динамической памяти и времени, поэтому единый алгоритм распределения динамической памяти будет подходящим только для подмножества приложений. Кроме того, удаление динамического выделения памяти из основной кодовой базы позволяет разработчикам приложений предоставлять свои собственные конкретные реализации, когда это необходимо.

Это становится особенно удобным при использовании возможностей некоторых архитектур, когда внешняя оперативная память может быть включена в адресное пространство микроконтроллера и использована для динамического размещения компонентов ядра.

Проще говоря, когда FreeRTOS требуется оперативная память, вместо вызова malloc() она вызывает pvPortMalloc(). Когда ОЗУ освобождается, вместо вызова free() ядро вызывает vPortFree(). pvPortMalloc() имеет тот же прототип, что и функция malloc() стандартной библиотеки Си, а vPortFree() имеет тот же прототип, что и функция free() стандартной библиотеки Си. pvPortMalloc() и vPortFree() являются общедоступными функциями, поэтому их также можно вызывать из кода приложения.

FreeRTOS поставляется с пятью примерами реализации pvPortMalloc() и vPortFree(). В этой главе мы рассмотрим только heap_1.c, heap_2.c, heap_3.c, heap_4.c. Последняя реализация - heap_5.c достаточно сильно зависит от особенностей архитектуры и применяемого компилятора и должна детально рассматриваться применительно к описываемым особенностям.

Все поставляемые примеры определены в исходных файлах и соответственно, все они расположены в каталоге FreeRTOS/Source/portable/MemMang.

Схема Heap_1

Небольшие проекты обычно обладают достаточным объёмом оперативной памяти для того, чтобы все планируемые объекты ядра операционной системы могли свободно разместиться в ней. Как правило, такие объекты создаются до запуска планировщика задач. В этом случае память хоть и выделяется динамически, но это происходит до того, как приложение начинает выполнять какие-либо функции в реальном времени, и память остается выделенной все время существования приложения. Это означает, что схема распределения памяти может и не учитывать какие-либо сложные моменты и проблемы возникающие в ходе выделения памяти. Например, детерминизм, фрагментация. Взамен модель предоставляет нам маленький размер кода и простоту реализации.

Проще говоря, Heap_1.c реализует очень упрощенную версию pvPortMalloc() и не реализует vPortFree() вообще. Приложения, которые никогда не удаляют задачу или другой объект ядра, потенциально могут использовать heap_1.

Вы наверняка знакомы с требованиями некоторых стандартов относительно кода критически важных с коммерческой точки зрения и безопасности систем. В большинстве таких систем крайне не рекомендуется использовать динамическое выделение памяти. Этот запрет на динамическое выделение памяти связан с тем, что этот процесс имеет ряд неопределенностей, вызванных фрагментацией памяти и неудачными выделениями. Heap_1 всегда является детерминированным и не может фрагментировать память.

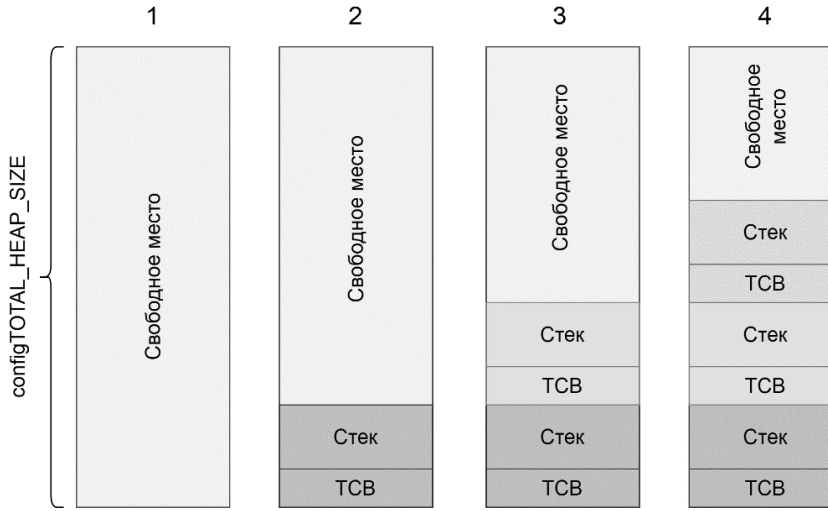


Рисунок 5. Схема heap_1. Работа с кучей.

Схема heap_1 чрезвычайно проста. В выделенной области памяти создается массив, который именуется кучей FreeRTOS. Этот массив делится на более мелкие фрагменты вызовом pvPortMalloc(). Общий размер массива в байтах определяется константой configTOTAL_HEAP_SIZE которая должна быть определена в конфигурационном файле FreeRTOSConfig.h.

Подобный подход имеет и очевидный недостаток – определение большого массива может привести к тому, что приложение будет использовать слишком большой объём памяти даже до того, когда фактически будут созданы компоненты ядра. Чтобы использовать эту память оптимально, следует на завершающих этапах отладки приложения измерить количество фактически потребляемого объёма памяти кучи FreeRTOS и, при необходимости, произвести корректировку выделяемого объёма памяти.

На рисунке 5: (1) показана вся куча FreeRTOS до того, как в ней будут размещены компоненты ядра. (2) – состояние кучи после того, как в ней будет размещена задача. Как видите для каждой задачи выделяются две области. В одной из них хранится TCB (Task Control Block) – структура, содержащая информацию о задаче и стек выделяемый самой задаче для хранения ее локальных переменных. (3) – созданы две задачи. (4) – состояние кучи FreeRTOS после того как были созданы три задачи.

Схема Heap_2

В современной идеологии FreeRTOS схема heap_2 является устаревшей. Эта схема включается в комплект поставки для обеспечения обратной совместимости с ранее созданными проектами, но она не рекомендуется для использования в новых проектах. Разработчики рекомендуют использовать heap_4 как более расширенную по функционалу версию управления кучей.

Heap_2.c точно так же оперирует с массивом размер которого определяется константой configTOTAL_HEAP_SIZE. Основное отличие от heap_1 заключается в том, что схема heap_2 использует алгоритм, позволяющий освобождать и повторно использовать память. Точно так же, как и в случае с heap_1, схема использует статически объявленный массив, поэтому память будет выделена и задействована даже в том случае, если она фактически не используется элементами ядра операционной системы.

Алгоритм, реализованный в схеме heap_2, позволяет функции pvPortMalloc() найти блок памяти наилучшим образом соответствующий запрашиваемому объёму памяти. В результате работы алгоритма будет использован блок памяти, который максимально близок к запрашиваемому объёму памяти. Например, представим, что куча содержит три блока свободной памяти с размерами 5, 25 и 100 байт соответственно. Функция pvPortMalloc() вызывается для выделения 20 байт. Наименьший свободный блок, в который поместится запрошенное число байтов, — это 25-байтовый блок, поэтому pvPortMalloc() разбивает 25-байтовый блок на один блок из 20 байт и один блок из 5 байт, прежде чем вернуть указатель на 20-байтовый блок. Новый 5-байтовый блок остается доступны для будущих вызовов pvPortMalloc().

В отличие от схемы heap_4, heap_2 не умеет объединять соседние свободные блоки в один блок большего размера, поэтому куча более подвержена фрагментации.

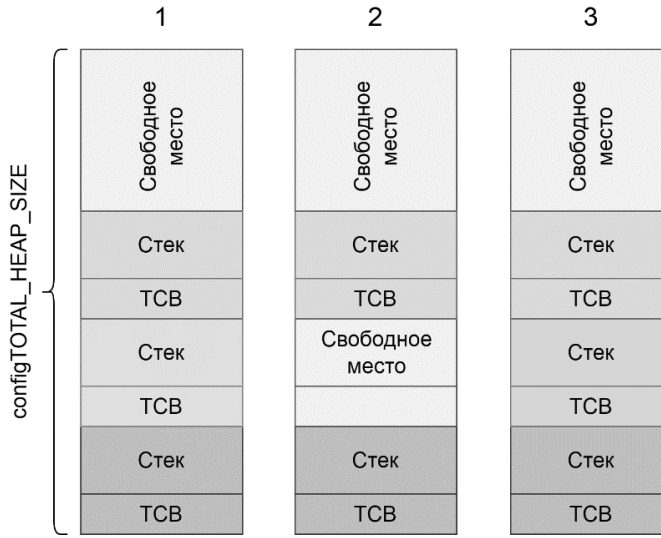


Рисунок 6. Схема heap_2.

На рисунке 6 схематично показан алгоритм, реализованный в heap_2, на примере задачи, которая была удалена и в последствии снова создана:

1. Показана куча после того, как были созданы три задачи.
2. Одна из задач была удалена, в результате чего, занимаемый блок памяти был освобожден.
3. Показана ситуация после создания другой задачи. Создание задачи привело к двум вызовам pvPortMalloc(): один для выделения нового TCB (Task Control Block), и один для выделения стека задач.

Следует отметить, что каждый TCB имеет одинаковый размер, поэтому алгоритм heap_2 гарантирует, что блок памяти, ранее выделенный для TCB удаленной задачи, повторно используется для выделения TCB новой задачи. В связи с тем, что размер стека, выделенного для вновь созданной задачи, идентичен размеру, выделенному для ранее удаленной задаче, блок ОЗУ, ранее выделенный для стека удаленной задачи, повторно используется для выделения стека новое задание.

Схема heap_2 не является детерминированной, но работает быстрее, чем большинство стандартных библиотечных реализаций malloc() и free().

Схема Heap_3

Схема heap_3.c использует стандартные библиотечные функции malloc() и free(), доступные в используемой реализации компилятора и языка. В связи с этим, размер кучи определяется конфигурацией компоновщика, и конфигурационный параметр configTOTAL_HEAP_SIZE не имеет значения.

В то же время, heap_3 не просто «оборачивает» вызов malloc() внутрь функции pvPortMalloc(). Схема делает вызовы malloc() и free() потокобезопасными. Это достигается за счет временной приостановки работы планировщика FreeRTOS.

По моему мнению, данная схема является худшим решением для прикладного применения в системах реального времени именно из-за приостановки работы планировщика и возникающей зависимости от времени выполнения библиотечных реализаций malloc() и free().

Схема Heap_4

Подобно ранее описанным схемам heap_1 и heap_2, heap_4 производит выделение памяти путем разделения статически выделенного массива на более мелкие блоки. Как и прежде размер массива определяется константой конфигурации - configTOTAL_HEAP_SIZE.

Схема heap_4 умеет объединять смежные свободные блоки в единый блок памяти, что очевидным образом способствует снижению риска фрагментации памяти, а используемый алгоритм подбора гарантирует, что pvPortMalloc() использует первый свободный блок памяти, достаточно большой для размещения запрошенного количества байтов. Например, рассмотрим сценарий, в котором куча содержит три блока свободной памяти, которые в порядке их появления в массиве составляют 5, 200 и 100 байт соответственно. Если произойдет вызов функции pvPortMalloc() с запросом на предоставление блока в 20 байт, то

первый свободный блок, в который поместится запрошенное количество байтов, — это 200-байтовый блок. В результате pvPortMalloc() разбивает 200-байтовый блок на один блок из 20 байт и один блок из 180 байт. Функция вернет указатель на 20-байтный блок, а новый 180-байтовый блок остается доступным для будущих вызовов pvPortMalloc().

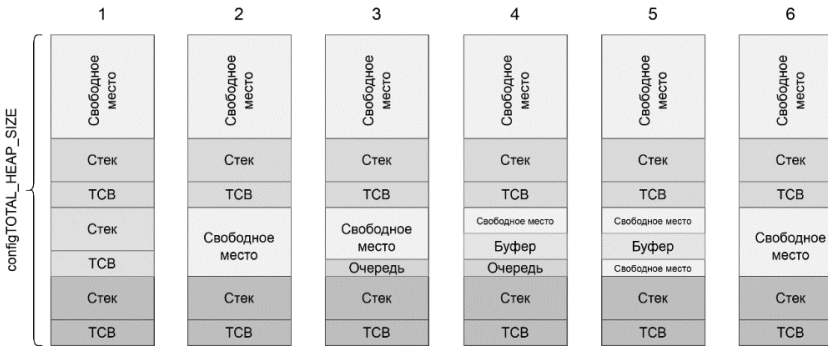


Рисунок 7. Схема heap_4.

На рисунке 7 представлены ситуации, отображающие реакцию схемы heap_4 на определенные изменения:

1. Показывает состояние кучи после того, как в ней была выделена память под три созданных задачи.
2. Показан массив после удаления одной из задач. Обратите внимание, что два блока памяти, высвободившиеся после удаления TCB и стека задачи, были объединены в единый массив.
3. Показывает ситуацию после того, как в результате вызова функции xQueueCreate(), была выделена память для размещения элементов очереди.
4. При выполнении прикладного программного обеспечения произошел вызов функции pvPortMalloc() напрямую, без использования API FreeRTOS. Блок, запрошенный приложением, был достаточно мал, чтобы поместиться в первый свободный блок.
5. Ситуация после того, как была удалена очередь.
6. Высвобождение выделенной прикладным программным обеспечением области памяти, используемой как буфер.

Несмотря на то, что схема heap_4 не является детерминированной, она работает быстрее, чем большинство стандартных библиотечных реализаций функций malloc() и free().

В момент написания этой книги схема heap_4 является самой популярной среди разработчиков. Она является схемой по умолчанию и в силу своей универсальности подходит для большинства приложений. Давайте рассмотрим некоторые необязательные параметры и настройки, позволяющие расширить возможности этой схемы.

Самой частой потребностью, возникающей у разработчиков, является желание разместить массив схемы heap_4, по определенному адресу памяти. Например, если используется несколько типов оперативной памяти в едином адресном пространстве микроконтроллера.

По умолчанию массив, используемый heap_4, объявлен внутри исходного файла heap_4.c, а его начальный адрес автоматически устанавливается компоновщиком в процессе сборки проекта. Однако, вы можете использовать константу конфигурации configAPPLICATION_ALLOCATED_HEAP определив ее в конфигурационном файле FreeRTOSConfig.h и присвоив ей значение 1. В этом случае массив должен быть объявлен пользовательским приложением, использующим FreeRTOS, и у программиста появляется возможность управлять размещением этого массива в оперативной памяти.

Если для configAPPLICATION_ALLOCATED_HEAP установлено значение 1, то в одном из исходных файлов приложения должен быть объявлен массив uint8_t с именем ucHeap и размерами, заданными параметром cconfigTOTAL_HEAP_SIZE. Например, давайте рассмотрим примеры объявления данного массива для компиляторов GCC и IAR.

Пример для GCC

```
uint8_t ucHeap[configTOTAL_HEAP_SIZE]
              __attribute__ ((section(".my_heap")));
```

Пример для IAR

```
uint8_t ucHeap[ configTOTAL_HEAP_SIZE ] @ 0x20000000;
```

Схема Heap_5

В рамках настоящей книги мы не будем глубоко вдаваться в подробности использования схемы heap_5 из-за того, что конкретная реализация работы данной схемы сильно зависит от особенностей используемого вами компилятора и компоновщика.

Алгоритм, реализованный в коде схемы heap_5 для выделения и освобождения памяти, полностью идентичен алгоритму, используемому heap_4. Однако в отличие от heap_4, heap_5 не ограничивается выделением памяти из одного статически объявленного массива; heap_5 может выделять память из нескольких областей памяти. Heap_5 полезен, когда ОЗУ, предоставленное системой, в которой работает FreeRTOS, не отображается как единый непрерывный блок адресного пространства микроконтроллера.

Чтобы выделение памяти в нескольких раздельных областях стало возможным heap_5 должна быть явно инициализирована перед вызовом pvPortMalloc(). Для инициализации используется API-функция vPortDefineHeapRegions(). Эта функция должна быть вызвана до создания каких-либо элементов ядра операционной системы (задач, очередей, семафоров и т. д.), т.е. до того, как потребуется выделение памяти.

Прототип функции vPortDefineHeapRegions() имеет следующий вид:

```
void vPortDefineHeapRegions( const HeapRegion_t *
                             const pxHeapRegions );
```

Как вы наверняка уже догадались vPortDefineHeapRegions() используется для указания начального адреса и размера каждой отдельной области памяти, которые вместе составляют общую память, используемую heap_5. Каждая отдельная область памяти описывается структурой типа HeapRegion_t. Описание всех доступных областей памяти передается в vPortDefineHeapRegions() в виде массива структур HeapRegion_t.

Единственным параметром, передаваемым в функцию vPortDefineHeapRegions(), является pxHeapRegions - указатель на начало массива структур HeapRegion_t. Каждая структура в массиве описывает

начальный адрес и длину области памяти, которая будет частью кучи при использовании heap_5.

```
typedef struct HeapRegion
{
   /* Начальный адрес блока памяти.*/
   uint8_t *pucStartAddress;
   /* Размер блока памяти в байтах. */
   size_t xSizeInBytes;
} HeapRegion_t;
```

В качестве примера рассмотрим пример, в котором имеются три независимых области памяти, обозначенные как RAM1 размером 32 килобайта, RAM2 и RAM3 размерами 32 килобайта каждый.

```
#define RAM1_START_ADDRESS ( ( uint8_t * ) 0x01000000 )
#define RAM1_SIZE ( 64 * 1024 )
#define RAM2_START_ADDRESS ( ( uint8_t * ) 0x01030000 )
#define RAM2_SIZE ( 32 * 1024 )
#define RAM3_START_ADDRESS ( ( uint8_t * ) 0x02000000 )
#define RAM3_SIZE ( 32 * 1024 )

/* Массив определений HeapRegion_t регионов. */
const HeapRegion_t xHeapRegions[] =
        {
           { RAM1_START_ADDRESS, RAM1_SIZE },
           { RAM2_START_ADDRESS, RAM2_SIZE },
           { RAM3_START_ADDRESS, RAM3_SIZE },
           { NULL, 0 } /* Конец массива. */
        };

int main( void )
{
   /* Инициализация схемы heap_5. */
   vPortDefineHeapRegions( xHeapRegions );

   /* Прочий код приложения... */
}
```

На рисунке 8 показано расположение выделяемых областей памяти в адресном пространстве.

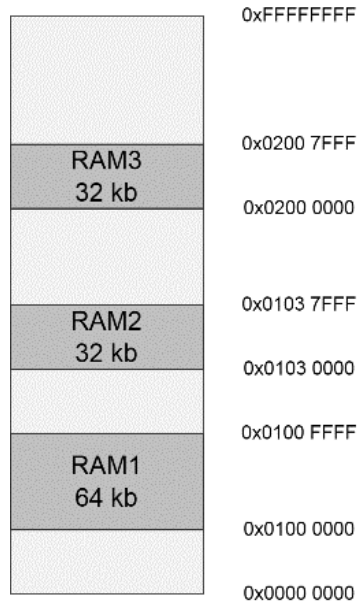


Рисунок 8. Выделяемые области памяти.

Функции работы с кучей

В процессе анализа (отладки) работы прикладного программного обеспечения порой требуется оценить какой объём кучи остается свободным. FreeRTOS API содержит две функции позволяющие в режиме реального времени получать информацию о использовании кучи.

```
size_t xPortGetFreeHeapSize( void );
```

Функция xPortGetFreeHeapSize() возвращает количество байт в куче, свободных на момент вызова функции. Это значение может быть использовано для оптимизации выделяемой под кучу памяти. Например, если вызов xPortGetFreeHeapSize() осуществленный после того, как все элементы ядра будут созданы возвращает значение 4096, то можно задуматься о том, что размер кучи может быть уменьшен.

Примите во внимание, что данная функция будет недоступна при использовании схемы heap_3.

Вторая функция более универсальна - xPortGetMinimumEverFreeHeapSize() возвращает минимальное количество нераспределенных байт, которые когда-либо существовали в куче с момента начала выполнения приложения FreeRTOS.

```
size_t xPortGetMinimumEverFreeHeapSize( void );
```

Универсальность данной функции заключается в том, что она производит замер свободного пространства кучи не в момент своего вызова, а показывает, насколько близко приложение когда-либо подходило к исчерпанию пространства в куче.

Функция xPortGetMinimumEverFreeHeapSize() доступна только при использовании схем heap_4 и heap_5.

Глава 4. Управление задачами

Это будет самая подробная глава книги, ведь представленные здесь концепции являются основополагающими для понимания того, как функционирует FreeRTOS и как ведут себя приложения, создаваемые с использованием этой операционной системы. Управление задачами, предоставление им доступа к ядру микроконтроллера является приоритетной и важнейшей функцией операционной системы. Для принятия решения о том, какой задаче предоставить доступ и когда, планировщик должен знать состояние каждой задачи, существующей в контексте операционной системы.

Создаваемое нами прикладное программное обеспечение может состоять из неопределенного множества задач. Вместе с тем, если микроконтроллер содержит только одно ядро, то в любой момент времени может выполняться только одна задача. Таким образом все задачи могут быть разделены на две категории «Running» (выполняется) и «Not Running» (не выполняется). Когда задача находится в состоянии «выполняется», процессор выполняет ее код. Когда задача «не выполняется» мы можем говорить о том, что ее состояние сохранено и сама задача ожидает того момента, когда планировщик решит, что он может предоставить ей доступ к процессору. Когда задача вновь получит этот доступ, ее выполнение будет продолжено с того же момента, в котором задача находилась в предыдущий раз, когда имела доступ к процессору. Т.е. с точки зрения задачи процесс переключения между задачами и разделения ресурсов процессора происходит незаметно.

В операционной системе FreeRTOS только планировщик может осуществлять переключение задач.

В главе 1 вопрос достаточного для каждой задачи времени уже обсуждался. Подавляющее большинство задач вовсе не нуждаются в том, чтобы иметь доступ к процессору постоянно. Всегда возможно определить индивидуальный ритм, в котором задача будет выполняться. Зачастую выполнение задачи может быть связано не только с некоторыми временными циклами, но и с наступлением каких-либо событий. Например, задача обрабатывающая поток входящих данных по интерфейсу UART. Не существует какой-либо работы для выполнения в те периоды времени, когда данные не поступают. И таких примеров может

быть множество - это и вывод информации на экран только в том случае, если имеется обновление данных; и задача, обрабатывающая события от органов управления.

Давайте рассмотрим этот вопрос еще раз. Если задачи, существующие в нашей системе, будут использовать все имеющееся в наличии доступное процессорное время, то распределение нагрузки возможно только за счет разделения времени, предлагаемого планировщиком. Т.е. задачи с наивысшим существующим приоритетом будут использовать все имеющееся время, и планировщик разделит его поровну между задачами. В то же время, задачи с более низким приоритетом попросту не получат времени для выполнения.

Если же мы реализуем ситуацию, в которой задачи самостоятельно выбирают темп выполнения, определяют временные интервалы, в течение которых они не нуждаются в доступе к ресурсам процессора, то у планировщика появляются свободные «тики», которые он может распределять для задач с низким приоритетом. Мы еще вернемся к этому вопросу немного позднее.

Для реализации концепта управления задач, при котором они могут быть приостановлены в ожидании какого-либо события в операционной системе FreeRTOS реализована расширенная модель состояний задач:

«Выполнение» Running — состояние, в котором задача фактически выполняется и использует ядро микроконтроллера. Соответственно, если микроконтроллер, на котором работает ОСРВ, имеет только одно ядро, то в любой момент времени может быть только одна задача в состоянии «Выполнение».

«Готова» Ready. В данном статусе находятся задачи, которые могут выполняться (они не находятся в состоянии «Заблокировано» или «Приостановлено»), но в настоящее время не выполняются, поскольку другая задача с таким же или более высоким приоритетом уже находится в состоянии «Выполняется» и использует процессор. Сразу после создания задача находится в состоянии «готова», и, если у планировщика нет более приоритетных задач, вновь созданная задача, получит отведенный ей квант времени при следующем же вызове планировщика.

«Заблокирована» Blocked — задача находится в заблокированном состоянии, если она в настоящее время ожидает события (событием может быть период времени или внешнее событие). Например, если

задача осуществит вызов функции vTaskDelay(), она будет заблокирована (переведена в состояние «Заблокировано») до истечения периода задержки — событие по времени. Задачи также могут блокироваться для ожидания очереди, семафора, группы событий, уведомления или события семафора. Задачи, находящиеся в состоянии «Заблокировано», обычно имеют период «тайм-аута», по истечении которого задача будет отключена и разблокирована (даже если событие, которого ожидала задача, не произошло). Задачи, находящиеся в состоянии «Заблокировано», не используют время обработки и не могут быть выбраны для перехода в состояние «Выполняется».

«Приостановлена» Suspended. Подобно задачам, находящимся в состоянии «Заблокировано», задачи в состоянии «Приостановлено» не могут быть выбраны для перехода в состояние «Выполнение», но для задач в состоянии «Приостановлено» время ожидания отсутствует. Вместо этого задачи входят в состояние Suspended или выходят из него только при явном указании на это через вызовы API vTaskSuspend() и xTaskResume() соответственно. Приостановленные задачи исключаются из планирования.

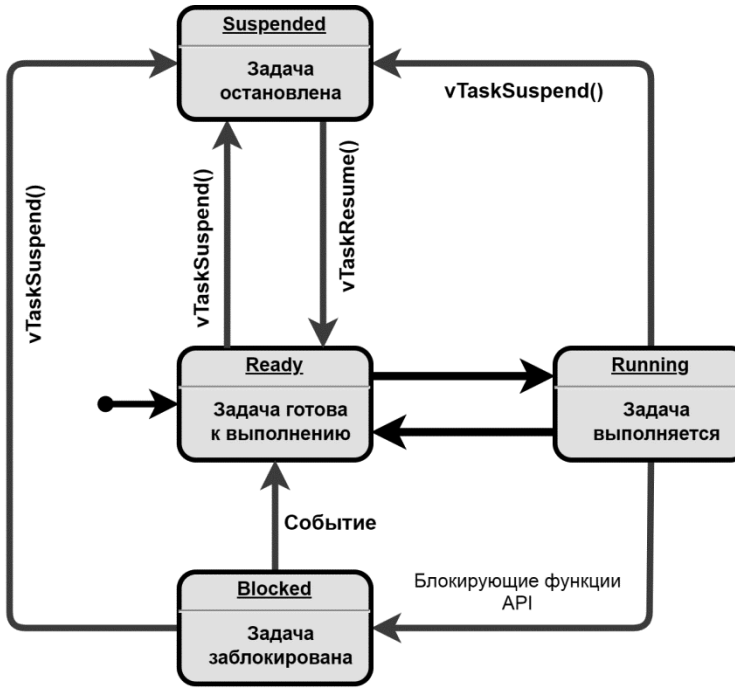


Рисунок 9. Состояния задач.

Приоритеты задач

Каждой задаче назначается приоритет. Первоначально, в момент создания задачи, приоритет определяется параметром uxPriority API-функции xTaskCreate(). После того, как планировщик FreeRTOS будет запущен приоритет задач можно изменять используя функцию vTaskPrioritySet().

Каждой задаче назначается приоритет от 0 до (configMAX_PRIORITIES - 1), где configMAX_PRIORITIES определяется в файле FreeRTOSConfig.h. Не стоит увлекаться и устанавливать слишком большое значение для константы configMAX_PRIORITIES. Количество приоритетов, определённых в системе, влияет на размер используемой оперативной памяти.

Чем меньше число, тем ниже приоритет задачи. Неактивная задача имеет нулевой приоритет (tskIDLE_PRIORITY). Планировщик FreeRTOS

гарантирует, что задачам в состоянии готовности или выполнения всегда будет отдаваться процессорное время (ЦП). Любое количество задач может иметь один и тот же приоритет. Если configUSE_TIME_SLICING не определён или если configUSE_TIME_SLICING установлен в 1, то задачи состояния готовности с равным приоритетом будут совместно использовать доступное время обработки с использованием схемы циклического планирования с временным разделением.

Планировщик FreeRTOS всегда гарантирует, что задачей с наивысшим приоритетом, которая может быть запущена, является задача, выбранная для перехода в состояние «Выполняется». Если может выполняться более одной задачи с одинаковым приоритетом, планировщик по очереди переводит каждую задачу в состояние «Выполняется» и из него.

Квантование времени

Если существует две и более задачи с одинаковым приоритетом ожидается, что, выполняя свои функции планировщик предоставит каждой из них сопоставимые отрезки времени для доступа к ядру микроконтроллера. Тем самым будет реализовано «квантование времени» его разбивка на равные промежутки. В начале кванта времени выбранная планировщиком задача будет получать доступ к процессору, переходя в состояние «выполняется», в конце кванта возвращаться в состояние «готова».

Для того, чтобы иметь возможность выбрать задачу для последующего запуска, планировщик должен получать управление. В большинстве операционных систем это происходит за счет использования периодических прерываний. Так называемых «tick interrupt».

Длина, размер кванта времени фактически устанавливается частотой с которой генерируются тиковые прерывания. Эта частота может быть установлена как константой configTICK_RATE_HZ определяемой в конфигурационном файле FreeRTOSConfig.h, так и в конфигураторе кода STM32CubeMX при создании проекта.

Рисунок 10. Основные настройки ОС в программе STM32CubeMX.

В некоторых функциях API операционной системы FreeRTOS требуется указать длительность или временной интервал в «тиках». Эти значения всегда необходимо указывать в виде кратных периодов.

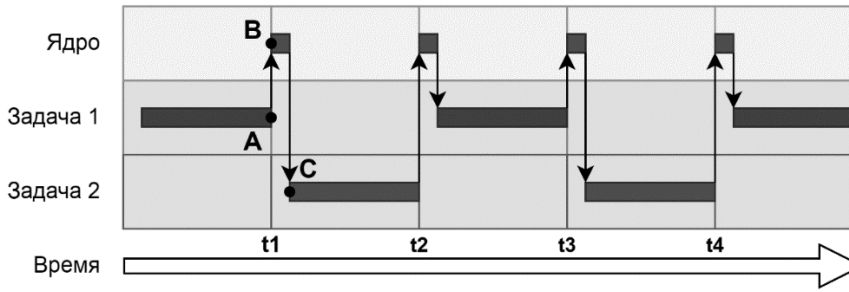


Рисунок 11. Квантование времени в реальной системе.

Давайте внимательно разберемся с тем, как происходит квантование времени в реальной системе. В примере показаны две задачи. Задача 1 и Задача 2. В интервалы времени t1 – t4 в системе происходят тиковые прерывания, являющиеся механизмом, запускающим выполнение планировщика. Планировщик, реализуя выбранную стратегию, распределяет доступное время между задачами. Обратите внимание, что на работу самого планировщика тоже расходуется малая часть процессорного времени. С тем, насколько много ресурсов расходуется на работу самого планировщика, мы обязательно разберемся немного позже.

По умолчанию configTICK_RATE_HZ = 1000, что соответствует времени тика равному 1 миллисекунде. Если вам потребуется изменить это значение и использовать частоту тиков отличную от значения по умолчанию, вы можете воспользоваться макросом pdMS_TO_TICKS(),

который преобразует время, указанное в миллисекундах, во время, указанное в тиках.

Реализация задачи

Задачи реализованы как функции Си, единственной их особенностью является их прототип:

```
void ATaskFunction( void *pvParameters );
```

Задача, реализуемая как функция Си должна возвращать параметр void и принимать параметр указателя void.

Каждая задача представляет собой небольшую «самостоятельную» программу. У задачи есть точка входа, есть собственный бесконечный цикл.

```
void vUserTaskFunction (void * pvParameters)
{
    // Обьявление локальных переменных
    // Однократно выполняемый фрагмент кода
    for( ;; )
    {
        // Код задачи
    }

    vTaskDelete (NULL);
}
```

Важно, чтобы функция, являющаяся воплощением задачи, не содержала оператора return. Если задача более не требуется она должна быть явно удалена вызовом функции vTaskDelete(). Представленный выше прототип задачи вызывает эту функцию, если бесконечный цикл for будет по каким-либо причинам прерван.

Очень удобным является тот факт, что одно объявление функции – задачи может использоваться для создания любого количества задач. Каждая создаваемая задача является отдельным экземпляром, имеет свой собственный стек, локальные переменные.

Создание задачи

Задачи создаются при помощи функции xTaskCreate(). Прототип данной функции выглядит следующим образом:

```
BaseType_t xTaskCreate(  TaskFunction_t pxTaskCode,
                         const char * const pcName,
                         const configSTACK_DEPTH_TYPE usStackDepth,
                         void * const pvParameters,
                         UBaseType_t uxPriority,
                         TaskHandle_t * const pxCreatedTask )
```

pvTaskCode — это указатель на функцию, реализующую задачу (по сути, просто имя функции).

pcName - имя задачи. Имя не используется операционной системой и указывается только для отладки.

usStackDepth - размер стека, выделяемого задаче. Параметр указывает количество слов, а не байт.

pvParameters - параметр, передаваемый функции воплощающей в себе задачу. По сути значение, присвоенное pvParameters, — это значение, переданное в задачу.

uxPriority - приоритет, с которым будет выполняться задача.

pxCreatedTask - передает дескриптор создаваемой задачи. В последствии его можно использовать для ссылки на задачу при различных вызовах функций API.

Существуют только два возвращаемых значения: pdPASS - задача создана успешно, и pdFAIL – указывающее на то, что задача не была создана. Последнее может свидетельствовать о том, что недостаточно памяти кучи, доступной для FreeRTOS.

Рассмотрим пример кода задачи:

```
void vTask1( void *pvParameters )
{
  const char *pcTaskName = "Task 1 is running\r\n";
```

```
  volatile uint32_t ul;
  for( ;; )
  {
     vPrintString( pcTaskName );
     for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
     {
        __NOP();
     }
  }
}
```

Пример создания задачи:

```
int main( void )
{
   xTaskCreate( vTask1, "Task 1", 1000, NULL, 1, NULL );

   vTaskStartScheduler();

   for( ;; );
}
```

Приведенные выше примеры справедливы, если вы используете операционную систему, имплементировав ее в проект самостоятельно. Если проект создается генератором кода STM32CubeMX, то будет интегрирована реализация FreeRTOS - CMSIS-RTOS адаптированная к особенностям архитектуры микроконтроллеров, работающих на ядре ARM Cortex. За счет этой адаптации некоторые функции могут выглядеть непривычно. Например, создание задачи, имплементированное в проект генератором кода STM32CubeMX, выглядит следующим образом:

```
osThreadDef(defaultTask, StartDefaultTask,
            osPriorityNormal, 0, 128);
defaultTaskHandle = osThreadCreate(osThread(defaultTask), NULL);
```

Блокировка задачи

Идея реализации «периодических» задач, задач, не нуждающихся в постоянном непрерывном доступе к ресурсам микроконтроллера и самостоятельно определяющих темп своего выполнения очень удобна. Прежде всего за счет того, что задача получает универсальный механизм контроля над темпом собственного выполнения. Основываясь на модели

состояния задач, реализованной во FreeRTOS, мы можем использовать статус «Заблокирована» (англ. Blocked) для того чтобы обозначить период времени, в течении которого задача не нуждается в доступе к ресурсам.

Переход задачи в заблокированное состояние происходит после вызова API-функции vTaskDelay(). Данная функция доступна только если константа INCLUDE_vTaskDelay определена в конфигурационном файле FreeRTOSConfig.h и имеет значение 1.

```
void vTaskDelay( TickType_t xTicksToDelay );
```

xTicksToDelay - количество тиковых прерываний, в течение которых вызывающая задача будет оставаться в состоянии «Заблокировано».

vTaskDelay() - это функция API предоставляемая FreeRTOS. Значение, передаваемое в качестве параметра, исчисляет время блокировки в «тиках». Реализацией CMSIS RTOS, которая распространяется вместе с генератором кода STM32CubeMX, предлагается аналогичная по функционалу функция osDelay(), принимающая в качестве параметра время блокировки определенное в миллисекундах.

```
osStatus osDelay (uint32_t millisec);
```

Это различие не имеет никакого значения vTaskDelay() при использовании частоты «тиков» по умолчанию, когда 1 тик = 1 миллисекунде, но должно учитываться при нестандартной частоте тиков.

Функции osDelay() и vTaskDelay() отсчитывают время относительно момента, когда они были вызваны. Их параметры определяют временной отрезок между вызовом функции, в результате чего задача была заблокирована и тем временем, когда задача вновь вернется в состояние готовности к выполнению.

Вместо этого параметры функции vTaskDelayUntil() определяют точное значение счетчика тиков, при котором вызывающая задача должна быть переведена из состояния Blocked в состояние Ready. vTaskDelayUntil() — это функция API, которую следует использовать, когда требуется фиксированный период выполнения (когда вы хотите, чтобы ваша задача выполнялась периодически с фиксированной частотой),

поскольку время разблокировки вызывающей задачи является абсолютным, а не относительным когда была вызвана функция.

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

pxPreviousWakeTime Исходя из того, что vTaskDelayUntil() используется для реализации задачи, которая выполняется периодически и с фиксированной частотой. Параметр pxPreviousWakeTime отражает время, когда задача в последний раз переходила из состояния «Заблокировано». Это время используется в качестве точки отсчета для определения времени выхода задачи из состояния блокировки.

xTimeIncrement Параметр указывает на какое значение в тиках задача должна быть переведена в заблокированное состояние. Можно использовать макрос pdMS_TO_TICKS() для преобразования времени, указанного в миллисекундах во время, указанное в тиках.

Основное различие между идеями заложенными в описанные выше функции vTaskDelay() и vTaskDelayUntil() на первый взгляд не столь очевидно. Обе эти функции могут быть использованы для реализации периодических задач. Однако задача, реализованная с использованием функции vTaskDelay(), не гарантирует, что частота выполнения будет строго фиксированной, поскольку время выхода задач из состояния Blocked зависит от времени, когда она вызвала vTaskDelay(). В то же время задача, вызвавшая функцию vTaskDelayUntil() вместо vTaskDelay(), не зависит от этой потенциальной проблемы, ведь ориентируется на абсолютное время.

```
void vTask01( void *pvParameters )
{
   char *pcTaskName;
   TickType_t xLastWakeTime;
   pcTaskName = ( char * ) pvParameters;
   xLastWakeTime = xTaskGetTickCount();
```

```
   for( ;; )
   {
      vPrintString( pcTaskName );
      vTaskDelayUntil( &xLastWakeTime, pdMS_TO_TICKS(250) );
   }
}
```

Блокирующие и не блокирующие задачи

Рассматривая концепцию организации процесса разделения времени между задачами, мы уже рассматривали два возможных сценария. В первом случае мы можем организовать задачи с одинаковым приоритетом, не предусматривая в их коде какой-либо функции, блокирующей их выполнение. В этом случае планировщик организует разделение имеющегося в его распоряжении доступного процессорного времени, разделив его между задачами. Во втором случае мы можем организовать задачи таким образом, чтобы они реализовывали модель периодической блокировки, направленной на поддержание необходимого темпа выполнения кода задач. Тем самым задача, переходя в состояние блокировки освобождает ресурсы для их распределения между другими задачами.

Попробуем провести эксперимент. Создадим три задачи. Задачи 1 и 2 будут работать в неблокирующем режиме. Не будут вызывать каких-либо функций API для перехода в состояние блокировки.

Задача 3 будет иметь небольшой «нагрузочный код», имитирующий полезную нагрузку, выполняемую периодически. Раз в 3 миллисекунды. Задача будет иметь более высокий, по сравнению с задачами 1 и 2 приоритет.

```
void StartTask01(void const * argument)
{
  for(;;)
  {
     __NOP();
  }
}

void StartTask02(void const * argument)
{
  for(;;)
  {
```

```
      __NOP();
  }
}

void StartTask03(void const * argument)
{
   TickType_t xLastWakeTime;
   const TickType_t xDelay3ms = pdMS_TO_TICKS( 3 );
   for(;;)
   {
        for (uint32_t x = 0; x < 0xFFe; x++) __NOP();
        vTaskDelayUntil( &xLastWakeTime, xDelay3ms );
   }
}
```

Запустим логический анализатор и посмотрим на процесс переключения задач. Как видно из диаграммы, задача 3 выполняется строго периодически и получает управление раз в 3 миллисекунды. На графике 5 приведены миллисекундные отсечки, совпадающие с началом очередного тика. Задачи 1 и 2 равномерно разделяют оставшееся, неиспользованное задачей 1, время.

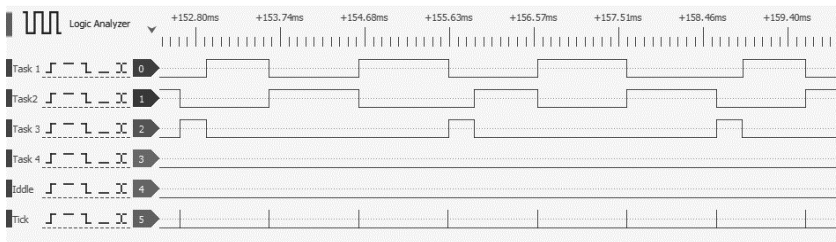


Рисунок 12. Задачи с различным приоритетом.

Задача простоя

Задача простоя (Idle Task) создаётся автоматически при запуске планировщика RTOS, это необходимо для того, чтобы гарантировать, что всегда существует хотя бы одна задача, которая может быть запущена – переведена в статус «выполняется». Для того, чтобы задача простоя не мешала и не использовала процессорное время при наличии любых других задач, теоретически способных выполняться, она создаётся с минимально возможным приоритетом.

Нужно заметить, что ничего не мешает разработчику создать любую другую задачу с нулевым приоритетом, в результате чего она будет разделять доступное время с задачей простоя. Однако реальных причин поступать таким образом мною найдено не было.

Обратите внимание, что задача простоя, помимо своей очевидной функции (утилизации свободного процессорного времени) имеет еще и важное значение для очистки ресурсов ядра операционной системы после удаления какой-либо задачи вызовом функции vTaskDelete(). Таким образом, если вы используете функцию удаления задач в своем программном обеспечении, то должны позаботиться и о том, чтобы Idle Task периодически имела время для выполнения и очистки мусора.

Задача простоя существует всегда, она является необходимым компонентом и было бы глупо не воспользоваться ее уникальными свойствами. При желании мы можем добавить собственный код в задачу простоя с помощью функции обратного вызова (Callback). Эта функция автоматически вызывается задачей бездействия один раз за итерацию цикла бездействующей задачи.

На практике, существует много примеров, которые могут быть реализованы при помощи встраивания пользовательского кода в задачу простоя:

- Выполнение различных фоновых задач;
- Мониторинг нагрузки. Функция обратного вызова задачи простоя получит управление только тогда, когда все задачи с более высоким приоритетом, не требуют времени; поэтому измерение количества времени, выделенного задаче бездействия, дает четкое представление о том, сколько времени остается свободным;
- Может использоваться для перевода микроконтроллера в режим пониженного энергопотребления. Обеспечивает простой и автоматический метод экономии энергии.

Существует ряд ограничений, действий, которые категорически нельзя выполнять в задаче простоя:

- Категорически запрещается использовать любые функции API операционной системы, способные перевести задачу в состояние блокировки. В этом случае в системе не останется ни одной задачи способной выполняться;

- Если создаваемое вами приложение использует API-функцию удаления задач vTaskDelete(), то необходимо гарантировать, что в течение разумного периода времени появится квант времени не занятый другими задачами, во время которого произойдет вызов задачи простоя. Это связано с тем, что задача Idle отвечает за очистку ресурсов ядра после удаления задачи.

Прототип функции обратного вызова задачи простоя:

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

В этой главе, я неоднократно пользовался функцией обратного вызова для того, чтобы показать на графиках время, остающееся невостребованным другими задачами. Для этого был использован весьма простой код:

```
void vApplicationIdleHook(void)
{
   GPIOA->BSRR = GPIO_PIN_4;
   __NOP();
   GPIOA->BSRR = (uint32_t)GPIO_PIN_4 << 16U;
}
```

Как упоминалось выше, одно из часто используемых применений этой функции – реализация кода, ведущего учет неиспользуемого процессорного времени. Это можно реализовать весьма простым кодом:

```
volatile uint32_t ulIdleCycleCount = 0UL;

void vApplicationIdleHook( void )
{
   ulIdleCycleCount++;
}
```

Чтобы использовать перехватчик простоя (в некоторой документации — перехватчик холостого хода) необходимо определить константу configUSE_IDLE_HOOK в конфигурационном файле FreeRTOSConfig.h, присвоив ей значение 1.

Практические эксперименты

Чтобы лучше разобраться в процессе функционирования задач и научиться точно прогнозировать результат проделаем небольшой эксперимент. В качестве стенда, с целью получения диаграмм использовалась отладочная плата DevEBox с микроконтроллером STM32F407VGT6 и логический анализатор DreamSourceLab.

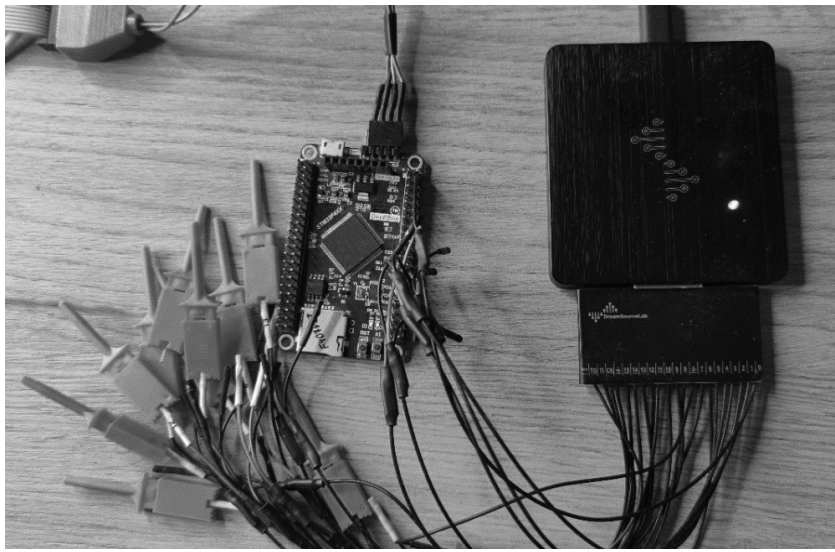


Рисунок 13. Используемый экспериментальный стенд.

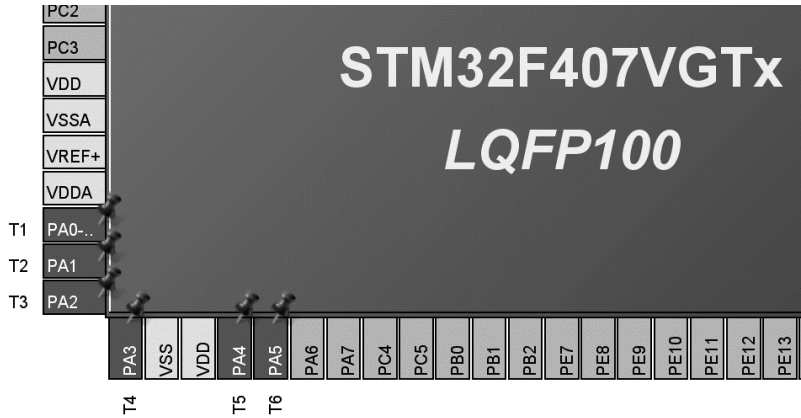


Рисунок 14. Настройка выводов микроконтроллера в STM32CubeMX.

Несколько выводов порта GPIOA были запрограммированы как GPIO_Output и подключены к логическому анализатору для отслеживания процесса переключения задач. Также были созданы 4 одинаковых задачи myTask01 – myTask04 на которых и будут проводиться все дальнейшие эксперименты.

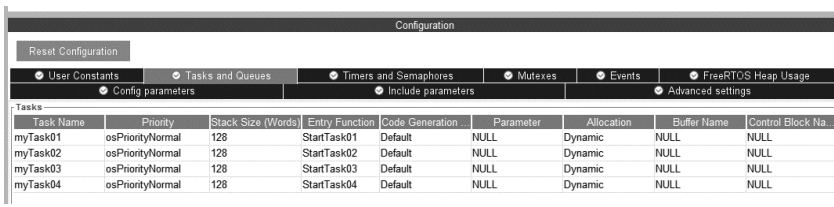


Рисунок 15. Настройка задач в STM32CubeMX.

Для визуализации времени, в течении которого будет выполняться каждая из задач были использованы стандартные макросы traceTASK_SWITCHED_IN() и traceTASK_SWITCHED_OUT(). Мы отдельно разберем макросы трассировки в отдельной главе, сейчас важным является тот факт, что эти макросы вызываются каждый раз, когда задача получает управление от планировщика и возвращает его. А значит, можно отобразить этот процесс на порты ввода вывода. Для этого в каждую задачу была добавлена метка.

```
void StartTask01(void const * argument)
{
  vTaskSetApplicationTaskTag(NULL,(void*)1);
  /* Infinite loop */
  for(;;)
  {
    osDelay(1);
  }
  /* USER CODE END 5 */
}
```

Теперь макрос трассировки может отличить одну задачу от другой.

Еще один порт ввода-вывода был назначен для того, чтобы показывать время каждого кванта времени используемого в прерывании системного таймера SysTick для управления планировщиком. Для этого обработчик прерывания SysTick был немного модифицирован:

```
void SysTick_Handler(void)
{
   HAL_IncTick();
   xPortSysTickHandler();

   GPIOA->BSRR = GPIO_PIN_5;
   __NOP();
   GPIOA->BSRR = (uint32_t)GPIO_PIN_5 << 16U;
}
```

Все готово к экспериментам.

Квант времени

Логично будет проверить тот факт, что планировщик получает управление каждую миллисекунду.

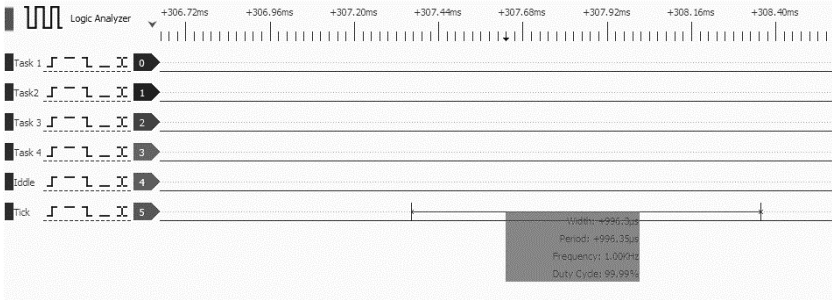


Рисунок 16. Тиковые прерывания.

На этом рисунке видно, что расстояние между двумя импульсами, соответствующими прерываниям от системного таймера, равняется 996 микросекунд. Спишем небольшую неточность на работу микроконтроллера без кварцевого резонатора.

Принимая во внимание тот факт, что мы создали 4 задачи с одинаковым приоритетом, логично предположить, что все доступное процессорное время будет поделено между ними поровну. Запустив логический анализатор, мы сможем проверить это предположение.

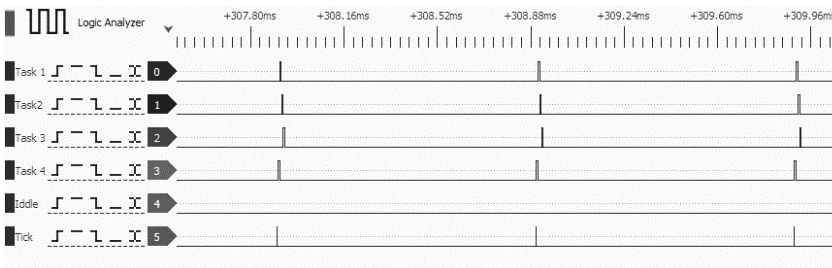


Рисунок 17. Последовательность выполнения задач.

Однако фактический график сильно отличается от ожидаемого. Что в увиденном вызывает вопросы?

Прежде всего можно заметить, что время, фактически израсходованное каждой из задач, сильно отличается от ожидаемого. В среднем каждая из задач получила только 6 микросекунд процессорного времени.

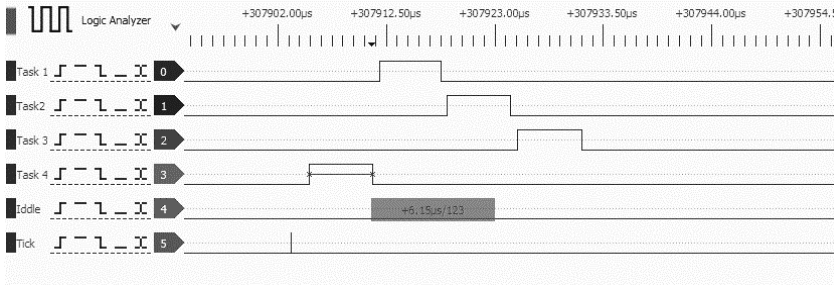


Рисунок 18. Последовательность выполнения задач.

Также нарушен порядок выполнения задач. После того, как одновременно с импульсом 5 канала анализатора – SysTick планировщик узнал о наступлении очередного кванта времени, он передал управление myTask04 вместо myTask01. Ведь задачи определены несколько в другой последовательности:

```
  /* Create the thread(s) */

  osThreadDef(myTask01, StartTask01, osPriorityNormal, 0, 128);
  myTask01Handle = osThreadCreate(osThread(myTask01), NULL);

   osThreadDef(myTask02, StartTask02, osPriorityNormal, 0, 128);
  myTask02Handle = osThreadCreate(osThread(myTask02), NULL);

  osThreadDef(myTask03, StartTask03, osPriorityNormal, 0, 128);
  myTask03Handle = osThreadCreate(osThread(myTask03), NULL);

  osThreadDef(myTask04, StartTask04, osPriorityNormal, 0, 128);
  myTask04Handle = osThreadCreate(osThread(myTask04), NULL);
```

Всему виной особенность, заложенная в алгоритм выбора задач планировщиком. Последняя созданная задача будет выполнена первой. И это остается верным до тех пор, пока задачи имеют одинаковый приоритет. Что будет если понизить приоритет последней созданной задачи?

```
  /* definition and creation of myTask04 */
  osThreadDef(myTask04, StartTask04, osPriorityBelowNormal, 0, 128);
  myTask04Handle = osThreadCreate(osThread(myTask04), NULL);
```

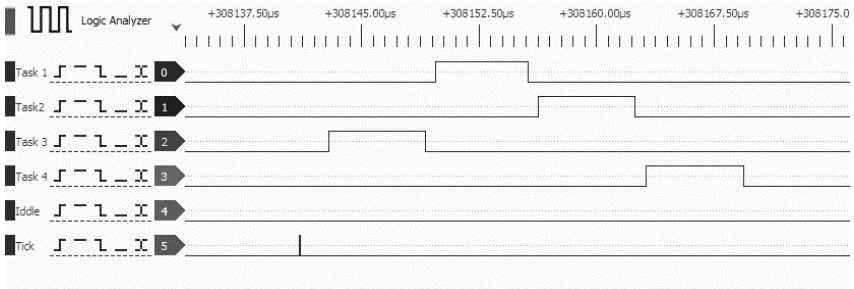


Рисунок 19. Пониженный приоритет Задачи 4.

Мы увидим, что задача myTask04 уступила свое место последней созданной задаче с более высоким приоритетом – myTask03 и заняла последнюю позицию в списке. Это очень удобный момент для того, чтобы провести ряд полезных измерений.

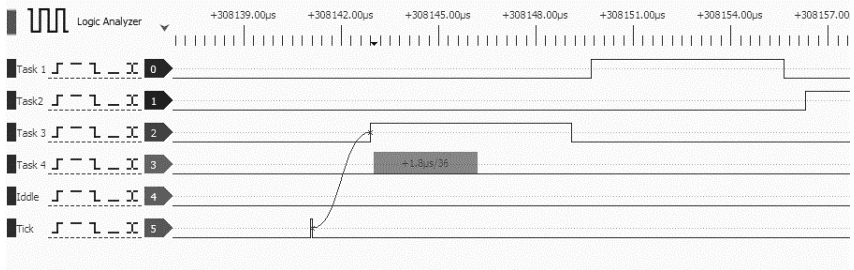


Рисунок 20. Время на принятие решения и передачу управления.

Из рисунка 20 можно увидеть, что планировщик тратит 1,8 микросекунды на то, чтобы определить какая из задач должна быть выполнена.

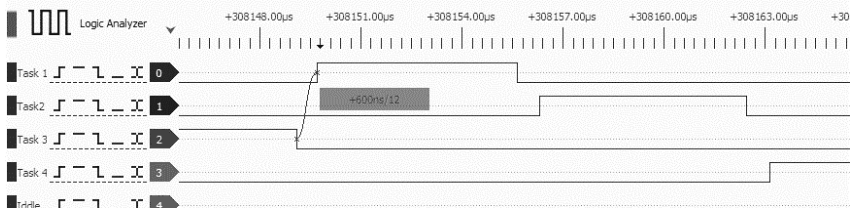


Рисунок 21. Время на переключение между задачами.

А также примерно 600 наносекунд на переключение между задачами.

На самом деле все эти измерения, хоть и являются в определенной мере примерными и ориентировочными, очень важны. Они дают представление о том, какие накладные расходы в виде процессорного времени будут затрачены не на фактическую работу задач, а на их обслуживание и функционирование компонент операционной системы.

Однако вернемся к ответу на вопрос, почему каждая задача получила так мало времени.

```
void StartTask02(void const * argument)
{
  for(;;)
  {
    osDelay(1);
  }
}
```

Наша задача состоит только из вызова одной единственной функции osDelay(). Эта функция по сути является «оберткой» для функции API операционной системы vTaskDelay() сообщающей планировщику, что вызывающая задача переходит в «заблокированное» состояние на ближайшую миллисекунду.

Этот факт и заставляет планировщик искать новую задачу и предоставлять ей процессорное время в освободившийся квант времени. Разумеется, эта ситуация имеет мало общего с реальностью, где задачи содержат код и вызовы функций, зачастую достаточно сильно нагружающие процессор. Дополним код задач имитацией полезной нагрузки. Например, поместив в код задачи длительный цикл.

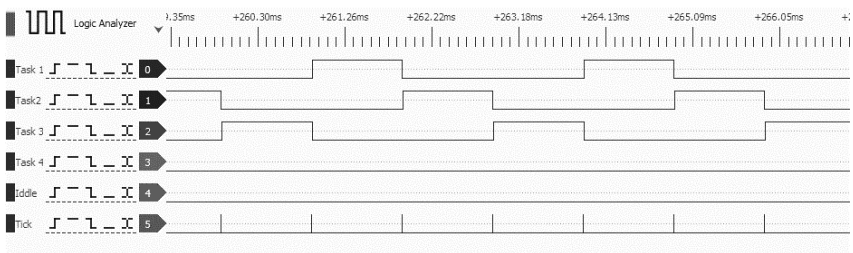


Рисунок 22. Задачи с имитацией реальной нагрузки.

Ситуация резко изменилась. Теперь каждая из задач myTask01 – myTask03 использует все отведенное им процессорное время – примерно 995 микросекунд. Но задача myTask04, имеющая более низкий приоритет вообще не имеет возможности выполняться, ведь задачи myTask01 – myTask03 полностью используют все доступное процессорное время.

Показанная ситуация является хорошим примером, но трудно представить, чтобы в реальной жизни существовали столь жесткие условия по использованию процессорного времени. Как правило, требования не столь жесткие.

Реальные задачи хоть и содержат код с различным временем выполнения, но, как правило, имеют более низкие требования к цикличности выполнения. К примеру, если одна из задач, является задачей опроса состояния клавиатуры то смысла делать это чаще, чем каждые 10 миллисекунд нет, даже с учетом функций программного подавления дребезга контактов. Это же применимо и к задаче опроса сенсоров, датчиков, задаче вывода информации на экран.

В качестве примера снизим нагрузку задач и выберем цикличность выполнения следующим образом: myTask01 – 10 миллисекунд; myTask02 – 25 миллисекунд; myTask03 – 60 миллисекунд и myTask04 – 2 миллисекунды.

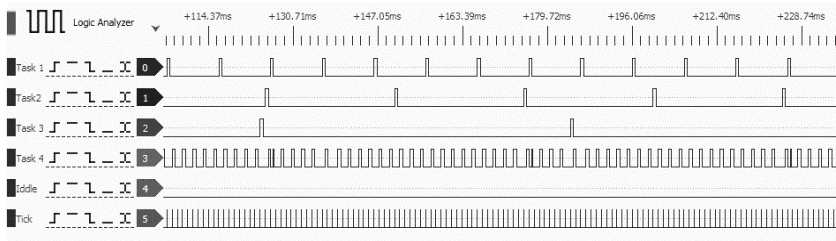


Рисунок 23. Циклически выполняемые задачи.

Как видно из рисунка это достаточно сильно меняет ситуацию. При сохранении низкого приоритета, установленного для задачи myTask04 она получает адекватное количество процессорного времени. Из графика можно заметить, что приоритеты по-прежнему работают. Равномерность циклов задач myTask01 – myTask03 не нарушается. Эта временная диаграмма весьма похожа на реальность и это вызывает закономерный

вопрос. Куда утилизируется свободное процессорное время и есть ли оно?

Дополним написанный ранее код

```
void vApplicationIdleHook(void)
{
   GPIOA->BSRR = GPIO_PIN_4;
   __NOP();
   GPIOA->BSRR = (uint32_t)GPIO_PIN_4 << 16U;
}
```

Функция vApplicationIdleHook() получает управление каждый раз, когда у операционной системы появляются свободные кванты времени. Тем самым происходит утилизация не используемого задачами времени.

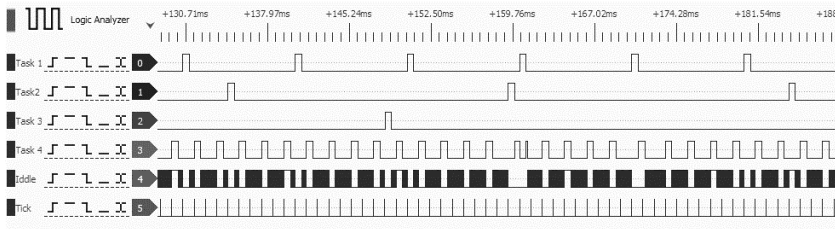


Рисунок 24. Утилизация времени в задачу простоя.

Обратите внимание на график задачи простоя (Iddle). Все закрашенные черным прямоугольники это неиспользуемое процессором время, время, которое было утилизировано вызовом функции vApplicationIdleHook(). И это очень важная информация.

Во-первых, подобный анализ говорит нам, что при желании мы можем организовать переход микроконтроллера в режим пониженного энергопотребления и тем самым экономить энергию. Во-вторых, мы явно имеем большое количество неиспользуемого процессорного времени, и мы можем снизить тактовую частоту микроконтроллера тем самым снизив и его энергопотребление. Какую из стратегий выбрать в качестве рабочей решать в каждом конкретном случае будет программист, мы лишь касаемся средства, позволяющего нам отслеживать такие процессы.

Функции управления приоритетами

Рассматривая параметры функции создания задач, было отмечено, что одним из передаваемых параметров является приоритет создаваемой задачи. После запуска планировщика, приоритет также может быть изменен вызовом API-функции vTaskPrioritySet(). Эту функцию можно использовать для изменения приоритета любой задачи. Обратите внимание, что API-функция vTaskPrioritySet() доступна, только если константа INCLUDE_vTaskPrioritySet определена в конфигурационном файле FreeRTOSConfig.h и имеет значение 1.

Прототип этой функции имеет следующий вид:

```
void vTaskPrioritySet(TaskHandle_t pxTask, UBaseType_t uxNewPriority);
```

pxTask дескриптор задачи, приоритет которой мы хотим изменить, можно изменить и приоритет задачи из которой осуществляется вызов функции, передав NULL вместо допустимого значения.

uxNewPriority Присваиваемый задаче приоритет, где максимально возможный приоритет - (configMAX_PRIORITIES — 1), где configMAX_PRIORITIES — это константа определенная в конфигурационном файле FreeRTOSConfig.h

Также существует функция, которая позволяет узнать текущий приоритет, установленный для задачи. Необходимо помнить, что функция uxTaskPriorityGet() доступна только в том случае, если константа INCLUDE_uxTaskPriorityGet определена в конфигурационном файле FreeRTOSConfig.h и имеет значение 1.

```
UBaseType_t uxTaskPriorityGet( TaskHandle_t pxTask );
```

pxTask дескриптор задачи, приоритет которой мы хотим определить, можно определить и приоритет задачи из которой осуществляется вызов функции, передав NULL вместо допустимого значения.

Возвращаемое значение - приоритет задачи

Удаление задач

Если возникает потребность в удалении какой-либо задачи, это можно сделать вызвав API-функцию vTaskDelete(). Задача, вызывающая vTaskDelete(), может удалить как саму себя, так и любую другую задачу. Обратите внимание, что API-функция vTaskDelete() доступна, только если для параметр INCLUDE_vTaskDelete определен в конфигурационном файле FreeRTOSConfig.h и имеет значение 1.

Удаленные задачи больше не существуют и не могут снова перейти в состояние «Выполняется». После того, как задача была удалена, Idle Task становится ответственной за освобождение памяти выделенной удаленной задаче.

Необходимо учитывать, что при удалении задачи очищается только память, выделенная для нее самим ядром операционной системы. Любая память или любой другой ресурс, выделенные самой задачей, должны быть освобождены явным образом.

```
void vTaskDelete( TaskHandle_t pxTaskToDelete );
```

pxTaskToDelete дескриптор задачи, которую мы хотим удалить, можно удалить и задачу из которой осуществляется вызов функции, передав NULL вместо допустимого значения.

Планировщик

С чего мы начинали разговор о многозадачности и статусах, в которых могут находиться задачи? Прежде всего с утверждения о том, что в микроконтроллерах, имеющих одно ядро, только одна задача может находиться в состоянии «выполняется» в каждый момент времени. Прочие задачи, если у них нет объективных причин не выполняться (не

находятся в состоянии блокировки и не приостановлены) находятся в статусе «Готовы» к выполнению. И именно из пула готовых к выполнению задач планировщику предстоит выбирать кандидата с наивысшим приоритетом для перехода в состояние выполнения.

Алгоритм, по которому планировщик выбирает задачу, которой будет предоставлен статус «выполняется» называется алгоритмом планирования и в операционной системе FreeRTOS есть возможность управлять этим алгоритмом.

Вы можете использовать три константы конфигурации configUSE_PREEMPTION, configUSE_TIME_SLICING и configUSE_TICKLESS_IDLE. Константы необходимо определить в конфигурационном файле FreeRTOSConfig.h и присвоить им значения 0 или 1 в зависимости от того, планируем мы их использование или нет. Стоит отметить, что последняя из перечисленных констант configUSE_TICKLESS_IDLE безусловно влияет на алгоритм планирования, однако это происходит косвенным образом. Константа configUSE_TICKLESS_IDLE предназначена для использования в режимах перехода микроконтроллера в режимы пониженного энергопотребления и нарушает работу тиковых прерываний.

Во всех возможных конфигурациях планировщик FreeRTOS обеспечит выбор задач с общим приоритетом для перехода в состояние выполнения по очереди. Эта политика «действуй по очереди» часто называется «циклическим планированием». Алгоритм циклического планирования не гарантирует, что время будет поровну распределено между задачами с одинаковым приоритетом, только то, что задачи в состоянии готовности с одинаковым приоритетом будут по очереди переходить в состояние «Выполняется».

Приоритетное упреждающее планирование

Воспользовавшись константами конфигурации, мы можем активировать самый распространенный и удобный для большинства приложений алгоритм планирования, который называется - «Упреждающее планирование с фиксированным приоритетом с разделением по времени». Именно этот алгоритм планирования показан во всех примерах данной книги и является алгоритмом, установленным как алгоритм по умолчанию.

```
#define configUSE_PREEMPTION                    1
#define configUSE_TIME_SLICING                  1
```

Что подразумевает его странное название, разберем его подробнее:

Фиксированный приоритет - Алгоритмы планирования, описанные как «фиксированный приоритет», не изменяют приоритет, назначенный запланированным задачам, но также не предотвращают изменение приоритета самих задач или других задач.

Упреждающий – возможно, стоило бы подобрать немного другой термин. Данный алгоритм действительно «упреждает» задачу. Если задача с более высоким приоритетом, чем выполняющаяся в данное время переходит в состояние «готово», упреждением будет перевод выполняющейся задачи из состояния «выполняется» в состояние «готова», чтобы позволить более приоритетной задаче перейти в состояние «выполняется».

Разделение времени - Разбивка по времени используется для распределения времени обработки между задачами с одинаковым приоритетом, даже если задачи явно не уступают или не переходят в состояние «Блокировано». Алгоритмы планирования, описанные как использующие «квант времени», будут выбирать новую задачу для перехода в состояние «Выполняется» в конце каждого кванта времени, если есть другие задачи в состоянии «Готово», которые имеют тот же приоритет, что и задача «Выполняется». Квант времени равен времени между двумя прерываниями по такту RTOS.

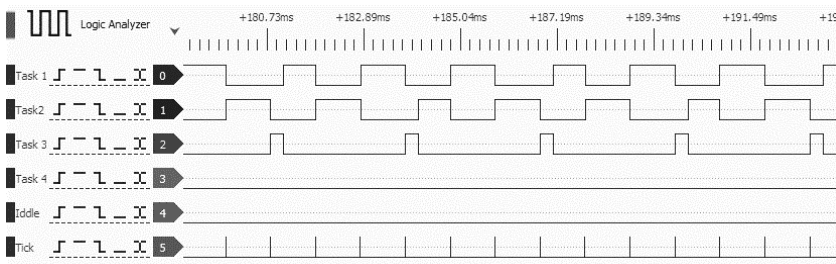


Рисунок 25. Приоритетное упреждающее планирование.

Упреждающее планирование с приоритетом

Упреждающее планирование с приоритетом без разделения по времени поддерживает те же алгоритмы выбора и вытеснения задач, что описаны в предыдущем разделе, но не использует разделение времени для его распределения между задачами с одинаковым приоритетом.

```
#define configUSE_PREEMPTION            1
#define configUSE_TIME_SLICING          0
```

Предыдущий алгоритм, используя квантование времени, оценивал и выбирал задачу для перехода в состояние «выполняется» каждый раз, когда происходило тиковое прерывание. Если же квантование времени и его распределение между задачами одного приоритета не используется, то планировщик выберет новую задачу для перехода в состояние «Выполняется» только в следующих случаях:

- Задача с более высоким приоритетом переходит в состояние «Готова».
- Задача в состоянии «Выполняется» переходит в состояние «Заблокировано» или «Приостановлено».

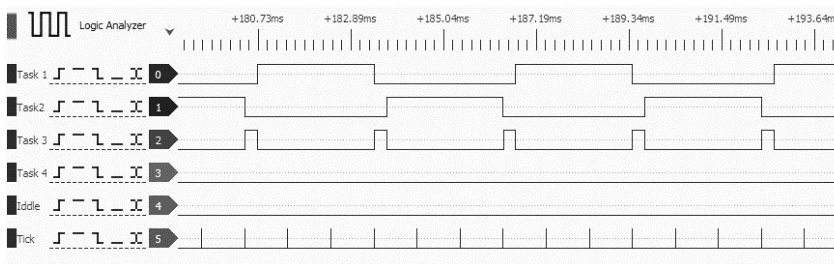


Рисунок 26. Упреждающее планирование.

Когда квантование времени не используется, переключается контекста задач происходит реже, чем когда используется квантование времени. Таким образом, отключение разделения времени приводит к сокращению накладных расходов планировщика на обработку. Одновременно это же приведет к тому, что задачи с одинаковым приоритетом получат совершенно разное время. По этой причине разработчики FreeRTOS относят данный метод планирования к методу для продвинутых пользователей.

Кооперативная многозадачность

До сих пор рассматривались исключительно примеры упреждающего планирования. Отчасти это объясняется тем, что весь объем вопросов по практической реализации механизма переключения задач целиком возлагается на планировщика. Ведь именно обеспечение совместного использования ресурсов (а в данном контексте и ядро микроконтроллера тоже совместный ресурс) заставило нас применить операционную систему в создаваемом проекте.

Между тем, FreeRTOS позволяет реализовать и механизм совместного планирования:

```
#define configUSE_PREEMPTION     0
#define configUSE_TIME_SLICING   0       // или 1
```

Когда используется кооперативный алгоритм работы планировщика, переключение контекста происходит только тогда, когда задача в состоянии выполнения переходит в состояние «Заблокировано» или когда задача в состоянии выполнения явно уступает (вручную запрашивает перепланирование) с помощью вызова taskYIELD(). Задачи никогда не вытесняются, поэтому разделение времени не может быть использовано.

Следует заметить, что у данного алгоритма тоже есть свои плюсы. Если мы используем алгоритмы упреждающего планирования, то должны прилагать усилия к исключению ситуаций, когда мы используем один и тот же ресурс одновременно в нескольких задачах. Например, две и более задачи не могут выводить данные на экран, не могут работать с периферийным устройством из-за реальной угрозы возникновения конфликтов. При кооперативном планировании сама задача решает, когда она будет готова перейти в состояние «готова» и прервать свое выполнение.

Иными словами, при использовании алгоритма кооперативного планирования, автор приложения контролирует, когда может произойти переключение на другую задачу. Тем самым, автор приложения должен гарантировать, что переключение на другую задачу не произойдет, пока ресурс находится в несогласованном состоянии.

Глава 5. Управление очередями

Что такое очередь? Формально очередь является хранилищем данных. Ведь по сути это действительно хранилище данных фиксированного размера, содержащее конечное число элементов. Максимальное количество элементов, которые может содержать очередь, называется ее «длиной», а длина и размер каждого элемента задаются при создании очереди.

Чаще всего очереди используются в качестве буферов First In First Out (FIFO), где данные записываются в конец (хвост) очереди и удаляются из начала (головы) очереди.

На рисунке 27 показаны две задачи и очередь. У каждой задачи есть локальные переменные. Задачи будут использовать очередь для обмена данными. После создания очередь пуста.

Рисунок 27. Пустая очередь.

Задача 1 в процессе работы присваивает переменной значение 9 и отправляет это значение в очередь. Значение становится первым элементом этой очереди. Очередь содержит один элемент.

Рисунок 28. Отправка данных в очередь.

Через некоторое время задача 1 присваивает новое значение локальной переменной и отправляет это значение в очередь. Это значение становится вторым элементом очереди. Теперь в очереди содержатся два элемента.

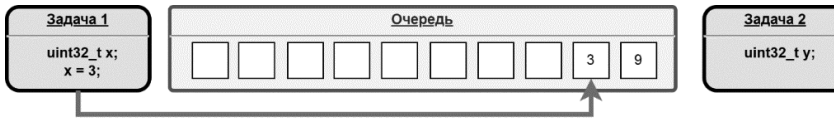


Рисунок 29. Отправка дополнительного элемента в очередь.

Предположим, что задача 2 получила управление и осуществила чтение из очереди. Ею будет прочитан первый элемент очереди и его значение присвоено локальной переменной.

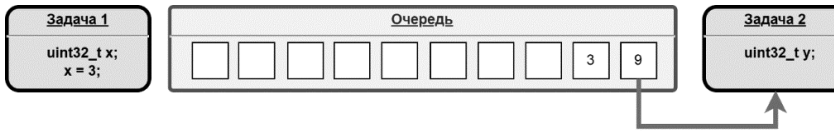


Рисунок 30. Чтение элемента из очереди.

Т.к. элемент очереди был прочитан, он будет удален из очереди и очередь сместится вперед. Таким образом очередь вновь будет содержать один элемент.

Рисунок 31. Сдвиг элементов в очереди.

Важной особенностью реализации механизма очередей в операционной системе FreeRTOS является тот факт, что операция постановки данных в очередь осуществляется путем копирования этих данных. В очередь помещается не ссылка на данные имеющиеся в распоряжении определенной задачи, а непосредственно сами данные.

Операционная система берет на себя все операции, связанные с выделением памяти, используемой для хранения элементов очереди.

Можно выделить несколько важных особенностей (свойств) очередей:

- Многозадачный доступ. Очереди — это самостоятельные объекты, любая задача или обработчик прерывания (ISR), знающие о существовании очереди, могут получить к ней доступ. Любое количество задач может помещать данные в одну и ту же очередь, и любое количество задач может читать из одной и той

же очереди. На практике очень часто очередь имеет несколько источников данных, но гораздо реже очередь имеет несколько «читателей».

- Таймаут при чтении из очереди. Когда задача пытается прочитать данные из очереди, она может дополнительно использовать время таймаута, отводимое операции чтения. Это время, в течение которого задача будет находиться в состоянии «Заблокирована» в ожидании доступности данных из очереди, если очередь из которой осуществляется чтение уже пуста. Задача, находящаяся в состоянии «Заблокирована» и ожидающая поступления данных из очереди, автоматически переходит в состояние «Готова», когда другая задача или обработчик прерывания помещает данные в очередь. Задача также будет автоматически переведена из состояния «Заблокирована» в состояние «Готова», если указанное время таймаута истечет до того, как данные станут доступны. Очереди могут иметь несколько «читателей» находящихся в ожидании данных из очереди, поэтому в одной очереди может быть заблокировано более одной задачи. Когда данные поступят будет разблокирована только одна задача. Разблокированная задача всегда будет задачей с наивысшим приоритетом, ожидающей данных. Если заблокированные задачи имеют одинаковый приоритет, то будет разблокирована задача, которая дольше всех ожидала данные.
- Таймаут отправителей данных. Так же, как и при чтении из очереди, задача может дополнительно использовать время таймаута, используемое при отправке данных в очередь. В этом случае время блокировки — это максимальное время, в течение которого задача будет находиться в состоянии «Заблокирована» в ожидании освобождения места в очереди, если очередь уже заполнена. Очереди могут иметь несколько источников, размещающих данные. В связи с этим может быть заблокировано более одной задачи, ожидающей завершения операции отправки. В этом случае только одна задача будет разблокирована, когда освободится место в очереди. Разблокированная задача всегда будет задачей с наивысшим приоритетом, ожидающей места. Если заблокированные задачи имеют одинаковый приоритет, то будет разблокирована задача, которая дольше всех ждала места.

- Группы очередей. Очереди можно группировать, что позволяет задачам переходить в состояние «Заблокирована» и ждать, пока данные станут доступными в любой из очередей группы.

Создание очереди

Очередь должна быть явно создана, прежде чем ее можно будет использовать. Для ссылки на очереди используются дескрипторы, которые являются переменными типа QueueHandle_t. Функция API xQueueCreate() создает очередь и возвращает QueueHandle_t, который ссылается на созданную ею очередь.

```
QueueHandle_t xQueueCreate( UBaseType_t uxQueueLength,
                            UBaseType_t uxItemSize );
```

uxQueueLength - максимальное количество элементов, которое может содержать создаваемая очередь в любое время.

uxItemSize - размер в байтах каждого элемента данных, который может храниться в очереди.

Возвращаемое значение - если NULL, то очередь не может быть создана, так как недостаточно памяти в куче для выделения структур данных очереди и области хранения. Значение отличное от NULL, указывает на то, что очередь успешно создана. Возвращенное значение должно быть сохранено как дескриптор созданной очереди.

Отправка данных в очередь

Несмотря на тот факт, что очередь является последовательностью элементов существуют API-функции позволяющие отправлять данные в «хвост» (конец) очереди - xQueueSendToBack() и в начало «голову»

очереди - xQueueSendToFront(). Помимо них существует и xQueueSend() полностью эквивалентная xQueueSendToBack().

Важно. Никогда не используйте функции xQueueSendToFront() или xQueueSendToBack() в обработчиках прерываний. Вместо них следует использовать безопасные функции xQueueSendToFrontFromISR() и xQueueSendToBackFromISR().

```
BaseType_t xQueueSendToFront( QueueHandle_t xQueue,
                              const void * pvItemToQueue,
                              TickType_t xTicksToWait );
```

и

```
BaseType_t xQueueSendToBack( QueueHandle_t xQueue,
                             const void * pvItemToQueue,
                             TickType_t xTicksToWait );
```

xQueue - дескриптор очереди, в которую отправляются данные. Дескриптор был возвращен xQueueCreate() использованной для создания очереди.

pvItemToQueue - указатель на копируемые в очередь данные. Размер каждого элемента, который может храниться в очереди, устанавливается при ее создании, именно это количество байт будет скопировано в область хранения очереди.

xTicksToWait - таймаут. Максимальное время, в течение которого задача должна оставаться в состоянии «Заблокирована» и ожидать освобождения места в очереди. Время указывается в тиках.

Возвращаемое значение - возможны два значения: pdPASS - данные были успешно отправлены в очередь и errQUEUE_FULL - данные не могут быть записаны в очередь.

Прототипы функций xQueueSendToFrontFromISR() и xQueueSendToBackFromISR(), предназначенные для безопасного

использования в обработчиках прерываний, а как следствие вне контекста операционной системы несколько отличаются:

```
BaseType_t xQueueSendToFrontFromISR( QueueHandle_t xQueue,
                    void *pvItemToQueue,
                    BaseType_t *pxHigherPriorityTaskWoken );
```

и

```
BaseType_t xQueueSendToBackFromISR( QueueHandle_t xQueue,
                    void *pvItemToQueue,
                    BaseType_t *pxHigherPriorityTaskWoken );
```

Как вы можете заметить, в них нет параметра связанного с возможностью блокировки задачи по таймауту. Это связано с тем, что обработчик прерывания выполняется вне контекста операционной системы. Но, выполнение операции с очередями может привести к изменению состояния задач, получающих данные из этой очереди. Как следствие статус задачи может измениться. Это и отслеживается с помощью параметра pxHigherPriorityTaskWoken. Подробнее о совместной работе FreeRTOS и системы обработки прерываний микроконтроллеров рассказывается в главе 6.

Получение данных

API-функция xQueueReceive() используется для получения (чтения) элемента очереди. Полученный элемент удаляется из очереди. Как и в случае с функциями, размещающими данные в очереди, запрещается использовать xQueueReceive() в коде обработчиков прерываний. Для этого существует специальная функция - xQueueReceiveFromISR().

```
BaseType_t xQueueReceive( QueueHandle_t xQueue,
                          void * const pvBuffer,
                          TickType_t xTicksToWait );
```

xQueue - дескриптор очереди, в которую отправляются данные. Дескриптор был возвращен xQueueCreate() использованной для создания очереди.

pvItemToQueue - указатель на переменную, в которую будут помещены читаемые из очереди данные.

xTicksToWait - таймаут. Максимальное время, в течение которого задача должна оставаться в состоянии «Заблокирована» и ожидать освобождения места в очереди. Время указывается в тиках.

Возвращаемое значение - возможны два значения: pdPASS - данные были успешно отправлены в очередь и errQUEUE_EMPTY - данных в очереди нет.

Нередко возникает ситуация, когда полезно знать количество сообщений, находящихся в очереди. В частности, это может быть использовано для изменения приоритета задачи, обрабатывающей очередь. API-функция uxQueueMessagesWaiting() используется для определения количества элементов находящихся в очереди.

```
UBaseType_t uxQueueMessagesWaiting( QueueHandle_t xQueue );
```

xQueue - дескриптор очереди.

Возвращаемое значение - количество элементов, находящихся в очереди.

Блокировка задач

Рассмотрим пример блокировки задачи ожиданием сообщения из очереди. Для иллюстрации определим две задачи Task01 и Task02 с нормальным приоритетом.

```
void StartTask01(void const * argument)
{
  for(;;)
  {
     __NOP();
  }
}
```

```
void StartTask02(void const * argument)
{
  for(;;)
  {
     __NOP();
  }
}
```

Доступное процессорное время будет разделено планировщиком между этими двумя задачами, имитирующими нагрузку. Как видно из кода примера задачи не вызывают каких-либо функций для искусственного переключения контекста.

Также, создадим периодически выполняющуюся задачу с высоким приоритетом. Task03 будет отправлять выполняться каждые 8 миллисекунд прерывая выполнение задач Task01 и Task02 и отправлять данные в очередь.

```
void StartTask03(void const * argument)
{
  uint32_t Item = 0;
  for(;;)
  {
     Item++;
     xQueueSend( myQueue01Handle, &Item, 10 );
     osDelay(8);
  }
}
```

И наконец, последняя задача – Task04 также имеет высокий приоритет и будет получать данные из очереди и копировать их в локальную переменную. Благодаря тому, что в функции чтения данных из очереди установлен максимальный период времени ожидания появления данных, большую часть времени задача Task04 будет находиться в состоянии «Заблокировано» ожидая появления данных несмотря на свой высокий приоритет.

```
void StartTask04(void const * argument)
{
  uint32_t Received;
  for(;;)
  {
    xQueueReceive( myQueue01Handle, &Received, portMAX_DELAY );
    for (uint32_t x = 0; x < 0x2FFe; x++) __NOP();
  }
```

```
}
```

Не трудно предположить график, который будет получен логическим анализатором.

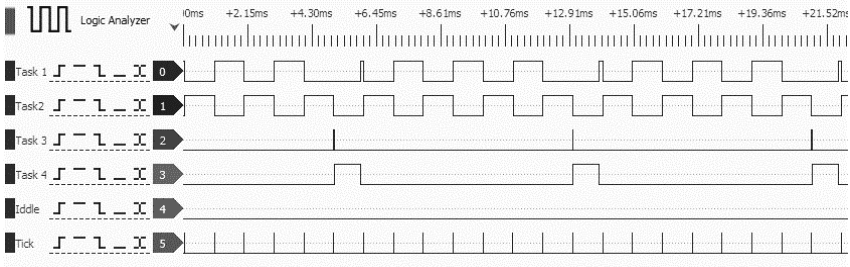


Рисунок 32. Блокировка задач очередями.

Рисунок 32 иллюстрирует поведение задач в точности соответствующее ожиданиям.

Немного усложним эксперимент и посмотрим на взаимные блокировки задач. В частности, рассмотрим случай, при котором в очередь с ограниченной длиной отправляются данные из нескольких источников и только одна задача осуществляет чтение. Логично предположить, что в скором времени очередь переполнится и начнут возникать таймауты, блокирующие работу отправителей. Рассмотрим это на примере.

Задачи 1, 2 и 3 имеют очень простой код. Они используют все доступное им процессорное время только для отправки данных в очередь.

```
void StartTask01(void const * argument)
{
  uint32_t Item = 1;
  for(;;)
  {
     xQueueSend( myQueue01Handle, &Item, portMAX_DELAY );
  }
}


void StartTask02(void const * argument)
{
```

```
  uint32_t Item = 2;
  for(;;)
  {
     xQueueSend( myQueue01Handle, &Item, portMAX_DELAY );
  }
}


void StartTask03(void const * argument)
{
  uint32_t Item = 3;
  for(;;)
  {
     xQueueSend( myQueue01Handle, &Item, portMAX_DELAY );
  }
}
```

Как следствие задачи очень быстро переполнят очередь. Читает данные из очереди только одна задача – Задача 4.

```
void StartTask04(void const * argument)
{
  uint32_t Received;
  for(;;)
  {
    xQueueReceive( myQueue01Handle, &Received, portMAX_DELAY );
    osDelay(1);
  }
}
```

Как видите, это периодическая задача. Она будет читать одно сообщение из очереди каждую миллисекунду. Давайте взглянем, как будут вести себя задачи через несколько миллисекунд после начала работы. Рисунок 33.

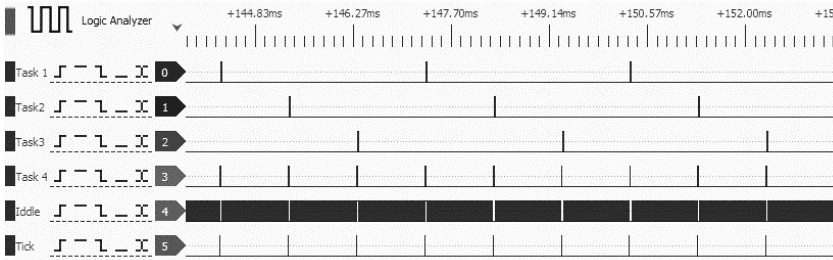


Рисунок 33. Блокировка отправителя данных в очередь.

Немного увеличим, чтобы была видна последовательность выполнения задач.

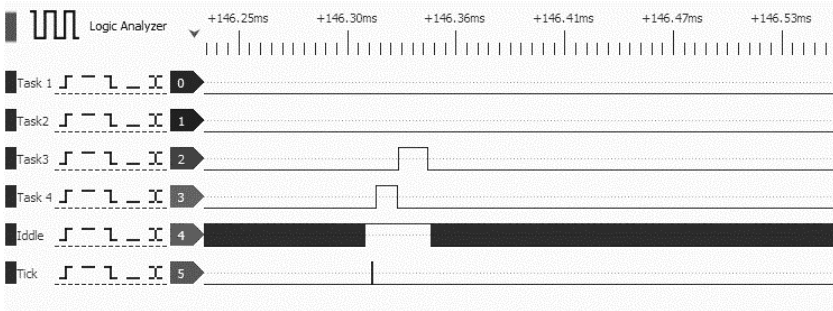


Рисунок 34. Последовательность выполнения задач при блокировке отправителя переполненной очередью.

Как видите задача 4 (осуществляющая чтение из очереди) предваряет задачу 3 (помещающую данные в очередь). Т.е. по сути деблокирует эту задачу. Мы еще вернемся к этому примеру в одной из глав.

Эти диаграммы показывают ситуацию уже после того, как очередь заполнилась и установилась описанная последовательность выполнения задач. А как же происходила работа приложения от момента его запуска и до того, как очередь оказалась заполнена.

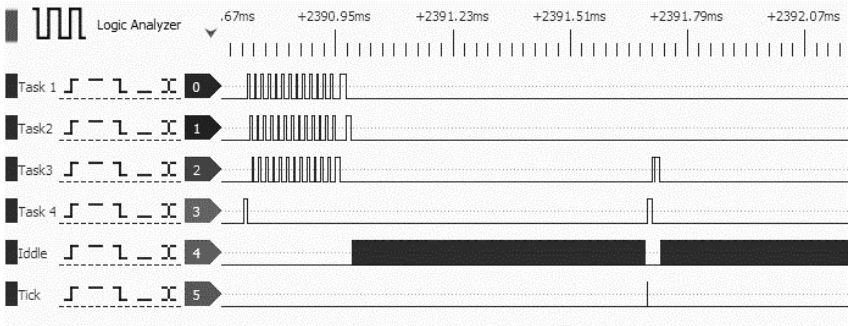


Рисунок 35. Фаза заполнения очереди.

Как видите Задачи 1, 2 и 3 активно выполнялись и конкурируя друг с другом наполняли очередь до тех пор, пока она не была заполнена.

Получение из нескольких источников

Существует огромное количество прикладных задач при решении которых существует необходимость сбора данных нескольких источников в единую задачу.

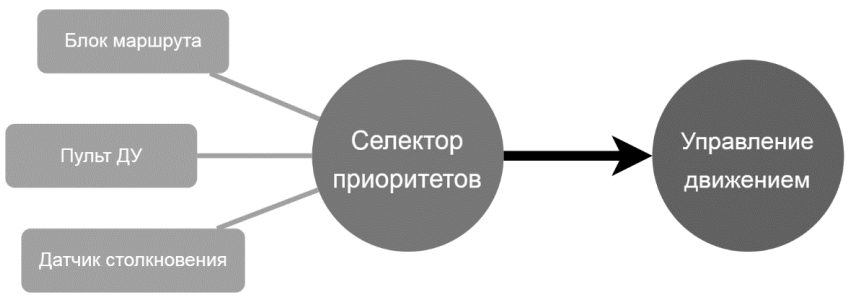


Рисунок 36. Получение данных из нескольких источников.

Даже если обратиться к организации работы робота-пылесоса. Для управления движением, формирование управляющих команд происходит отдельным алгоритмом, так называемым селектором приоритетов. Этот программный алгоритм принимает решение, что из возможных действий является приоритетным для робота. Какое должно быть выполнено в первую очередь, а какое может немного подождать. В своей работе,

селектор приоритетов опирается на информацию о возможных действиях, предоставляемую отдельными функциональными задачами. Например, задача обработки ситуаций, связанных со столкновением, может предложить последовательность действий, которую необходимо предпринять для обхода препятствия на пути робота. Задача обслуживания аппаратной части пульта дистанционного управления, может получить какие-то команды от пользователя, а блок следования по маршруту, может формировать команды, призванные вести робота по заранее намеченному маршруту.

Такой подход предполагает, что в очередь, связывающую задачи, формирующие поведение и задачу принимающую решение, могут попадать данные разного назначения. Например, от пульта дистанционного управления может поступить команда об увеличении скорости, а алгоритм навигации может потребовать осуществить поворот на 12 градусов. Мы можем организовать передачу и данных и описания данных в единую очередь благодаря использованию структур:

```
// Возможные команды
typedef enum
{
   eSpeed,
   eDirection,
   eAction
} DataDescription_t;

// Структура данных помещаемых в очередь
typedef struct
{
   uint8_t ucValue;
   DataDescription_t eDataDescription;
} Data_t;
```

Теперь каждая задача может помещать данные в единую очередь и в специальном поле структуры Data_t описывать, для регулирования какого параметра предназначены эти данные, что именно они описывают.

```
static void vReceiverTask( void *pvParameters )
{
  Data_t xReceivedStructure;
  BaseType_t xStatus;

  for( ;; )
```

```
  {
        xStatus = xQueueReceive( xQueue, &xReceivedStructure, 0 );
        if( xStatus == pdPASS )
        {
          switch (xReceivedStructure.eDataDescription)
          {
             case eSpeed:
                // Регулируем скорость
                uint8_t CurrentSpeed = xReceivedStructure.ucValue;
                break;
             case eDirection:
                // Изменить курс
                uint8_t CurrentDirection = xReceivedStructure.ucValue;
                break;
             case eAction:
                // Произвести действие
                uint8_t CurrentAction = xReceivedStructure.ucValue;
                break;
          }
        }
        else
        {
          // Очередь пуста
        }
  }
}
```

Такой подход позволяет добиться хорошей организации логики, принимающей данные из очереди задачи.

Данные переменной длины

В связи с тем, что память для всех элементов очереди выделяется в момент ее создания, использование очередей для передачи данных большого объема не кажется слишком хорошей идеей. Если у нас есть потребность в передаче больших объёмов данных, то предпочтительнее использовать очередь для передачи указателя на данные, а не копировать сами данные. Вместе с тем при отправке указателей в качестве элементов очереди необходимо соблюдать осторожность и проверить, что:

- Две и более задач не изменяют один и тот же участок памяти одновременно и не предпринимают каких-либо действий, способных привести к повреждению данных. В идеале только

задаче-отправителю должен быть разрешен доступ к памяти до тех пор, пока указатель на память не будет поставлен в очередь, и только задаче-получателю должен быть разрешен доступ к памяти после получения указателя из очереди.

- Только одна задача отвечает за освобождение памяти. Никакие задачи не должны пытаться получить доступ к памяти после ее освобождения. Указатель никогда не должен использоваться для доступа к данным, выделенным в стеке задач.

Рассмотрим такой подход на примере. Прежде всего нам необходимо создать очередь способную содержать указатели.

```
/* Обьявляем переменную типа QueueHandle_t дескриптор очереди. */
QueueHandle_t xPointerQueue;
/* Создаем очередь на 5 элементов. */
xPointerQueue = xQueueCreate( 5, sizeof( char * ) );
```

Следующим этапом нам необходима задача выделяющая буфер в памяти. Эта же задача заполняет буфер информацией и отправляет ссылку на него в очередь.

```
/* Задача создающая и формирующая буфер, отправляющая ссылку на буфер в очередь */
void vStringSendingTask( void *pvParameters )
{
  char *pcStrToSend;
  const size_t xMaxStrLength = 50;
  BaseType_t xStrNumber = 0;

  for( ;; )
  {
    pcStrToSend = (char *)prvGetBuffer(xMaxStrLength);
    sprintf(pcStrToSend, xMaxStrLength, "String %d\r\n", xStrNumber);
    xStrNumber++;
    xQueueSend( xPointerQueue, &pcStrToSend, portMAX_DELAY );
  }
}
```

Теперь нам необходима задача, которая помимо очевидного – получения данных из очереди и проведения над ними операций будет уметь и освобождать не использующийся буфер.

```
/* Задача получает данные из очереди,
производит над ними операции и очищает (освобождает) буфер. */
```

```
void vStringReceivingTask( void *pvParameters )
{
   char *pcReceivedString;
   for( ;; )
   {
      xQueueReceive( xPointerQueue, &pcReceivedString, portMAX_DELAY );
      /* Операции над полученными данными */
      prvReleaseBuffer( pcReceivedString );
   }
}
```

С точки зрения использования памяти такой подход очень рационален. Он позволяет обойтись без копирования и излишнего дублирования больших объемов данных.

Схожий подход может быть применен и в том случае, если мы имеем дело с данными размер которых динамически меняется. В большинстве литературы по FreeRTOS в качестве такого примера приводится та или иная реализация кода, оперирующего с IP стеком. В какой-то мере этот пример является компиляцией из отправки в очередь структуры данных и отправки в очередь ссылки на данные. Рассмотрим классический пример использованный разработчиками FreeRTOS для иллюстрации возможного подхода. В рассматриваемом примере используется одна очередь для получения данных разных типов от трех независимых источников данных. Этот пример практически использован в реализации стека FreeRTOS+TCP TCP/IP.

Стек TCP/IP, работающий в своей собственной задаче, должен обрабатывать события из множества различных источников. Различные типы событий связаны с различными типами и длинами данных. Все события, происходящие вне задачи TCP/IP, описываются структурой типа IPStackEvent_t и отправляются задаче TCP/IP в очереди.

```
typedef enum
{
   /* Интерфейс не активен, требуется переподключение. */
   eNetworkDownEvent = 0,
   eNetworkRxEvent, /* Получен пакет, событие. */
   eTCPAcceptEvent, /* Вызвана функция FreeRTOS_accept() */
} eIPEvent_t;

/* Структура описывающая события отправляемые задаче TCP/IP. */
typedef struct IP_TASK_COMMANDS
{
   eIPEvent_t eEventType;
```

```
   void *pvData;
} IPStackEvent_t;
```

Таким образом структура IPStackEvent_t содержит в себе элемент pvData и он может использоваться как для непосредственного хранения данных, так и для хранения указателя на буфер.

Разберемся с примерами событий и связанными с ними данными. Например, eNetworkRxEvent означает что из сети получен пакет данных. Данные, полученные из сети, должны быть переданы задаче TCP/IP с использованием структуры типа IPStackEvent_t. Элементу eEventType структуры присваивается значение eNetworkRxEvent, а элемент pvData структуры используется для указания на буфер, содержащий полученные данные.

```
void vSendRxDataToTheTCPTask( NetworkBufferDescriptor_t *pxRxedData )
{
   IPStackEvent_t xEventStruct;
   xEventStruct.eEventType = eNetworkRxEvent;
   xEventStruct.pvData = ( void * ) pxRxedData;

   xQueueSend( xEventToIPStackQueue, &xEventStruct, portMAX_DELAY );
}
```

Следующее событие – eTCPAcceptEvent указывает на то, что сокет должен принимать или ожидать подключения от клиента. События Accept отправляются из задачи, вызвавшей FreeRTOS_accept(), в задачу TCP/IP с использованием структуры типа IPStackEvent_t. Элементу eEventType структуры присваивается значение eTCPAcceptEvent, а элементу pvData структуры присваивается дескриптор сокета, принимающего соединение.

```
void vSendAcceptRequestToTheTCPTask( Socket_t xSocket )
{
   IPStackEvent_t xEventStruct;
   xEventStruct.eEventType = eTCPAcceptEvent;
   xEventStruct.pvData = ( void * ) xSocket;

   xQueueSend( xEventToIPStackQueue, &xEventStruct, portMAX_DELAY );
}
```

Завершающее этот пример событие – eNetworkDownEvent генерируется тогда, когда требуется подключение или повторное подключение к сети. События отключения сети отправляются из сетевого

интерфейса в задачу TCP/IP с использованием структуры типа IPStackEvent_t. Для элемента eEventType структуры установлено значение eNetworkDownEvent. События отключения сети не связаны с какими-либо данными, поэтому элемент структуры pvData не используется.

```
void vSendNetworkDownEventToTheTCPTask( Socket_t xSocket )
{
   IPStackEvent_t xEventStruct;
   xEventStruct.eEventType = eNetworkDownEvent;
   xEventStruct.pvData = NULL;

   xQueueSend( xEventToIPStackQueue, &xEventStruct, portMAX_DELAY );
}
```

Изучая код, который получает и обрабатывает события в задаче TCP/IP можно увидеть, что элемент eEventType структур IPStackEvent_t, полученных из очереди, используется для определения того, как следует интерпретировать элемент pvData. Этот подход уже был продемонстрирован при описании метода получения данных из нескольких источников.

```
IPStackEvent_t xReceivedEvent;
xReceivedEvent.eEventType = eNoEvent;
xQueueReceive( xNetworkEventQueue, &xReceivedEvent, xNextIPSleep );
switch( xReceivedEvent.eEventType )
{
   case eNetworkDownEvent :
     prvProcessNetworkDownEvent();
     break;
   case eNetworkRxEvent:
     prvHandleEthernetPacket((NetworkBufferDescriptor_t *)
                             (xReceivedEvent.pvData));
     break;
   case eTCPAcceptEvent:
     xSocket = ( FreeRTOS_Socket_t * ) ( xReceivedEvent.pvData );
     xTCPCheckNewClient( pxSocket );
     break;
}
```

Проблема использования очередей

Несмотря на тот факт, что очереди являются очень удобным и для многих программистов привычным инструментом есть и определенные

недостатки о которых необходимо помнить. Например, в главе 6 мы будем разбираться с тем, как организовать совместную работу системы обработки прерываний и операционной системы. Там в качестве примера будут разбираться некоторые вопросы переноса обработки из прерываний в задачу операционной системы. Нет смысла предварять идеи следующей главы, но давайте посмотрим на то, насколько на самом деле быстро работает очередь.

Проведем небольшой эксперимент. Создадим 3 задачи. В теле задач пропишем небольшой код весьма отдаленно имитирующий реальную загрузку задач.

```
void StartTask01(void const * argument)
{
  uint32_t Item = 1;
  for(;;)
  {
    for (uint32_t x = 0; x < 0x2FF; x++) __NOP();
    osDelay(1);
  }
}

void StartTask02(void const * argument)
{
  for(;;)
  {
    for (uint32_t x = 0; x < 0x2FF; x++) __NOP();
    osDelay(1);
  }
}

void StartTask03(void const * argument)
{
  uint32_t Received;
  for(;;)
  {
    xQueueReceive( myQueue01Handle, &Received, portMAX_DELAY );
    for (uint32_t x = 0; x < 0xFF; x++) __NOP();
  }
}
```

Задачи 1 и 2 просто содержат код осуществляющий несколько итераций цикла для создания иллюзии нагрузки и выполнения какой-то полезной работы. А задача 3 получает информацию из очереди. Отправителем информации может являться любой обработчик прерывания. В нашем случае это будет прерывание от таймера. Таймер

TIM3 будет формировать прерывания с частотой 19200 раз в секунду имитируя для нашего примера обычный UART работающий на скорости 19200 бод.

```
void TIM3_IRQHandler(void)
{
  uint32_t Item = 1;
  BaseType_t pxHigherPriorityTaskWoken;

  HAL_TIM_IRQHandler(&htim3);

  pxHigherPriorityTaskWoken = pdFALSE;
  xQueueSendFromISR(myQueue01Handle, &Item, &pxHigherPriorityTaskWoken);
  GPIOA->BSRR = GPIO_PIN_3;
  xQueueSendFromISR(myQueue01Handle, &Item, &pxHigherPriorityTaskWoken);
  GPIOA->BSRR = (uint32_t)GPIO_PIN_3 << 16U;
}
```

Как видите, наш обработчик весьма минималистичен. Он просто отправляет данные в очередь. Вслед за этим происходит разблокировка задачи 3 и обработка поступивших данных.

Уверен, большинство читателей уже догадываются, что при этом сценарии мы увидим самое начало проблем.

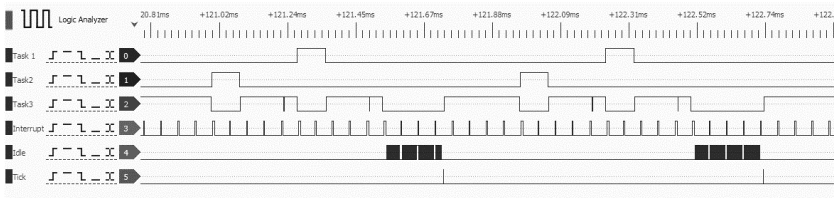


Рисунок 37. Потенциальная проблема использования очередей.

Посмотрите внимательно на график, полученный логическим анализатором. Основная часть времени работает задача 3 принимающая данные из очереди.

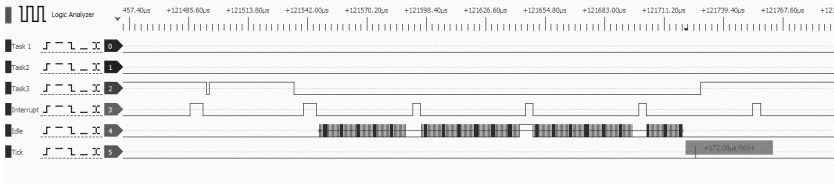


Рисунок 38. Возникновение дефицита времени.

Из этого графика видно, что в конце каждого системного тика остается меньше 170 микросекунд свободного времени, которое утилизируется в задаче простоя. И это при тактовой частоте 72 МГц. А что будет, если мы попробуем имитировать скорость 38400 бод?

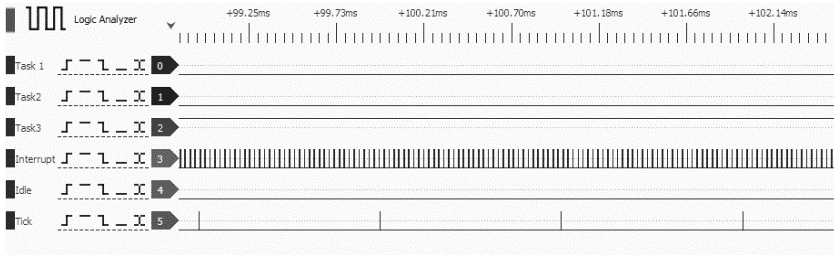


Рисунок 39. Дефицит времени для выполнения задач 1 и 2.

Нет, это не сбой в работе логического анализатора. Задачи 1 и 2 остановлены для них не хватает процессорного времени. Задача 3 расходует 100% доступного процессорного времени для обработки поступающих данных. Система не работоспособна.

Я специально поместил этот пусть упрощенный, но вполне жизненный пример в одну из первых глав книги. Нужно понимать, что это не недостаток операционной системы, это недостаток дизайна прикладного программного обеспечения, созданного с применением операционной системы. Есть быстрые механизмы обмена информацией, есть медленные. Мы должны изучить имеющиеся у нас возможности и понять их достоинства и недостатки, для того, чтобы создавать поистине удачное программное обеспечение. К счастью, очереди, являются самым медленным объектом, позволяющим производить обмен данными.

Глава 6. Обработка прерываний

Самое время вспомнить о тех понятиях, которые мы рассматривали в теоретической части главы 2. Пытаясь дать определение тому, что характеризует операционную систему реального времени было упомянуто, что сам термин реального времени применительно к операционной системе это способность операционной системы обеспечить требуемый уровень сервиса в определённый промежуток времени, а именно обеспечить гарантированное время реакции на внешние события. Очевидно, говоря о микроконтроллерах, внешние события — это прерывания от оборудования.

Иными словами, операционные системы реального времени должны предпринимать некоторые действия в ответ на возникающие события. Таким событием может быть прерывание от часов реального времени, какого-либо периферийного блока, например, SPI, UART, внешней линии прерывания EXTI. Операционная система должна надлежащим образом обслуживать эти события, в том числе и тогда, когда события приходят из нескольких источников. При этом, отдельной задачей является реализация некоторой стратегии обработки этих событий.

В документации от разработчиков FreeRTOS отмечается, что ОСРВ не навязывает разработчику приложения какой-либо конкретной стратегии которой следовало бы придерживаться при обработке событий. Однако, ОСРВ предоставляет функции, которые позволяют реализовать выбранную стратегию простым и удобным способом. На самом деле разработчику, использующему FreeRTOS и испытывающему необходимость реализовать обработку прерываний, необходимо определиться только с тем, какая часть обработки прерывания должна выполняться в самом обработчике, а какая часть может реализована вне обработчика.

Для понимания принципов и идей настоящей главы важно различать приоритет задачи и приоритет прерывания:

- Задача — это программная функция, не связанная с аппаратным обеспечением, на котором работает FreeRTOS. Приоритет задачи назначается в программном обеспечении автором

приложения, а программный алгоритм (планировщик) решает, какая задача будет находиться в состоянии выполнения.

- Обработчик прерывания (ISR) – код в том же программном обеспечении, но является аппаратной функцией, поскольку аппаратное обеспечение определяет, какая функция обработки прерывания будет выполняться и когда она будет выполняться. Задачи будут выполняться только тогда, когда нет запущенных ISR, поэтому прерывание с самым низким приоритетом прервет задачу с наивысшим приоритетом, и у задачи нет возможности упредить ISR.

Функции API и обработчики прерываний

Реализуя приложения с использованием FreeRTOS часто возникает необходимость обратиться к API-функциям ОС в обработчике прерывания (ISR). Необходимо помнить, что многие функции API FreeRTOS выполняют действия, недопустимые внутри ISR. Представим ситуацию, когда во время выполнения задачи «А» произошло прерывание и управление было передано обработчику прерывания. Если внутри обработчика мы вызовем функцию переводящую задачу в состояние «Заблокировано», то какая задача должна быть заблокирована? Ведь если какая-либо функция API FreeRTOS вызывается из обработчика прерывания, значит она вызывается не из задачи. Т.е. у нее нет вызывающей задачи.

FreeRTOS решает эту проблему, предоставляя две версии некоторых функций API; одна версия для использования из задач, другая для использования в коде обработчика прерываний. К именам функций, предназначенных для использования в ISR, добавляется «FromISR».

Нужно заметить, что такой подход не уникален и похожее разделение функций API иногда применяется. Как правило, такой подход позволяет добиться нескольких положительных эффектов. Так наличие отдельного API для использования в обработчиках прерывания (вне контроля, осуществляемого планировщиком операционной системы) позволяет сделать и код задачи более эффективным и код ISR — более эффективным. Что это значит для нас? Прежде всего тот факт, что код функций API не перегружен функционалом, позволяющим ему определять, вызывается эта функция из обработчика прерывания или из задачи. На втором месте – вопросы переносимости кода операционной

системы, ведь не все архитектуры позволяют с легкостью определить контекста выполнения, а значит потребуется еще большее усложнение кода.

Между тем, необходимость использования двух различных функций, написанных только для использования в контексте операционной системы и в обработчиках прерываний, показывают еще одну «проблему». Такой функционал дает возможность переключения контекста во время выполнения кода обработчика прерывания. Иными словами, задача, выполняемая при выходе из обработчика прерывания, может отличаться от задачи, которая выполнялась при входе в обработчик прерывания. Сам факт возникновения прерывания прервал одну задачу, но вернул управление другой задаче. Это вызвано тем, что некоторые функции API FreeRTOS могут переводить задачу из состояния «Заблокировано» в состояние «Готово». Яркий пример такого изменения состояния задач – использование функции xQueueSendToBack(), которая разблокирует задачу, если она находилась в состоянии «Заблокировано», ожидая, пока данные в очереди станут доступными.

Рассматривая алгоритмы, реализуемые планировщиком, мы могли заметить, что если приоритет задачи, разблокированной функцией API FreeRTOS, выше приоритета задачи, находящейся в состоянии «Выполняется», то в соответствии с политикой планирования должно произойти переключение на задачу с более высоким приоритетом.

Важным является не сам факт данного переключения, а то, когда произойдет фактическое переключение на задачу с более высоким приоритетом. И это напрямую зависит от контекста, из которого вызывается функция API:

- Если функция API была вызвана из задачи. Поведение зависит от того, какое значение установлено для константы configUSE_PREEMPTION. Если установлено значение 1, то переключение на задачу с более высоким приоритетом происходит автоматически внутри функции API, то есть до выхода из функции API.
- Если функция API была вызвана из прерывания, то переключение на задачу с более высоким приоритетом не произойдет автоматически внутри прерывания. Вместо этого устанавливается переменная, информирующая автора

приложения о том, что следует выполнить переключение контекста.

Для этих целей все функции, предназначенные для использования внутри обработчиков прерывания («FromISR») имеют параметр-указатель, pxHigherPriorityTaskWoken, который используется для этой цели. Если необходимо выполнить переключение контекста, функция установит для *pxHigherPriorityTaskWoken значение pdTRUE. Перед использованием, переменная, на которую указывает pxHigherPriorityTaskWoken, должна быть инициализирована значением pdFALSE.

Это даст возможность автору создаваемого программного обеспечения самостоятельно решить, что он пожелает сделать: или запросить переключение контекста или оставить задачу с высоким приоритетом в состоянии готовности до следующего запуска планировщика, что в самом худшем случае произойдет во время следующего тикового прерывания.

Обратите внимание, что функции API могут устанавливать *pxHighPriorityTaskWoken только в значение pdTRUE. И если внутри обработчика прерывания вызывается несколько функций, способных изменить статус какой-либо задачи, следует инициализировать переменную значением pdFALSE прежде чем она будет использована первый раз.

Существует несколько причин, по которым переключение контекста не происходит автоматически внутри функции API, разработанной для использования в обработчиках прерываний:

1. Требуется избегать ненужных (излишних) переключений контекста. В качестве удачного примера можно привести пример, когда необходимо принять несколько байт по интерфейсу UART и только затем выполнить обработку. Ведь прерывание от UART генерируются при получении каждого нового байта, а переключение контекста задачи может требоваться только после получения полной строки.
2. Попытка контроля за последовательностью управления. Прерывания могут происходить спорадически и в непредсказуемое время. С большой вероятностью автор приложения может пожелать заблокировать непредсказуемое переключение при выполнении определенных фрагментов кода.

Обратите внимание, что использование параметра pxHigherPriorityTaskWoken не является обязательным. Если у вас нет потребности контроля за переключением контекста, установите для pxHigherPriorityTaskWoken значение NULL.

Макросы portYIELD_FROM_ISR() и portEND_SWITCHING_ISR()

В этом разделе мы познакомимся с двумя макросами, описанными в файле порта, а значит специфическими для архитектуры микроконтроллера. Макросы применяются для запроса переключения контекста.

taskYIELD() — используется для переключения контекста из задачи. portYIELD_FROM_ISR() и portEND_SWITCHING_ISR() являются версиями макроса taskYIELD() предназначенными для вызова из кода обработчика прерывания. portYIELD_FROM_ISR() и portEND_SWITCHING_ISR() используются одинаково и делают одно и то же.

```
portEND_SWITCHING_ISR( xHigherPriorityTaskWoken );
```

и

```
portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
```

Параметр xHigherPriorityTaskWoken, полученный от функции способной вызвать переключение контекста, можно использовать непосредственно в качестве параметра в вызове portYIELD_FROM_ISR().

Если параметр portYIELD_FROM_ISR() xHigherPriorityTaskWoken равен pdFALSE (ноль), то переключение контекста не запрашивается, и макрос не действует. Если параметр portYIELD_FROM_ISR() xHigherPriorityTaskWoken не равен pdFALSE, то запрашивается переключение контекста, и задача в состоянии выполнения может измениться. Прерывание всегда будет возвращаться к задаче в состоянии выполнения, даже если задача в состоянии выполнения изменилась во время выполнения прерывания.

Отложенная обработка прерываний

Среди программистов, работающих в embedded считается хорошей практикой делать обработчики прерываний как можно короче. У этого есть вполне очевидное объяснение – прерывание может возникнуть в любой момент времени, соответственно, и обработчик прерывания может быть вызван в любой момент времени. Для того, чтобы система в целом сохранила возможность быстрой и бесперебойной реакции на прерывания, крайне важно, чтобы обработчик прерывания выполнялся очень быстро и возобновлял управление прерванного кода. Это не только будет являться гарантией того, что у системы будет достаточно времени для реакции на новые события и прерывания, но и функционирование основного кода приложения будет продолжено.

Однако, реакция на некоторые прерывания может потребовать достаточно большую подготовительную работу или сопряжено со взаимодействием с медленными интерфейсами или периферийными устройствами. Хорошим примером являются различного рода прерывания от сетевых устройств. Они сопряжены с большим объёмом обрабатываемой или получаемой информации.

И если вы уже имеете определенный опыт программирования систем, построенных на прерываниях, используя FreeRTOS, вы столкнетесь с новыми особенностями, которые придется учитывать и принимать во внимание:

- даже если задачам назначен очень высокий приоритет, они будут выполняться только в том случае, если аппаратное обеспечение не обслуживает прерывания;
- прерывания могут нарушить (добавить временной «джиттер») как время начала, так и время выполнения задачи;
- в зависимости от архитектуры, на которой работает FreeRTOS, может быть невозможно принять какие-либо новые прерывания или, по крайней мере, подмножество новых прерываний во время выполнения текущего обработчика;
- разработчик приложения должен учитывать последствия и защищаться от одновременного доступа задачи и обработчика прерывания к таким ресурсам, как переменные, периферийные устройства и буферы памяти;

- несмотря на то, что большинство портов FreeRTOS допускают вложение прерываний, вложение прерываний может повысить сложность и снизить предсказуемость. Чем короче прерывание, тем меньше вероятность его вложенности.

Вместе с тем, концепция операционной системы предлагает и вполне изящный выход из ситуации «перегруженности» обработки прерываний. Обработчик прерывания должен записывать (сохранять) причину прерывания и очищать прерывание. Любая другая обработка, необходимая в рамках возникшего прерывания, может выполняться в задаче, позволяя процедуре обслуживания прерывания завершиться так быстро, как это практически возможно. Это и называется «отложенной обработкой прерывания», потому что обработка, вызванная прерыванием, «откладывается» и переносится из обработчика в задачу.

Этот стиль позволяет разработчику приложения устанавливать приоритет обработки по отношению к другим задачам и использовать все доступные функции API FreeRTOS.

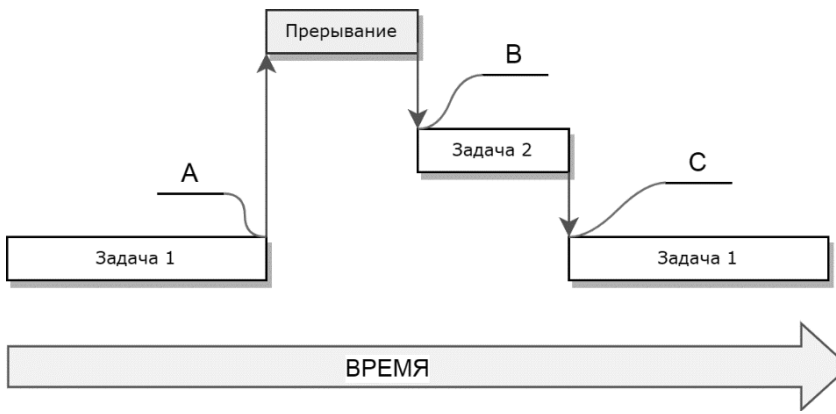


Рисунок 40. Отложенная обработка прерывания.

Рассмотри предлагаемую концепцию. Задача 2 – содержит код обработчика прерывания. Задача существует в контексте операционной системы и находится в состоянии блокировки т.к. основную массу времени нет необходимости обрабатывать прерывание и осуществлять какие-либо связанные с ним действия. В момент «A» происходит аппаратное прерывание. Выполнявшаяся «Задача 1» будет прервана и управление передано обработчику прерывания. Если следовать

предлагаемому концепту, обработчик сохраняет данные полученные в обработчике прерывания от инициировавшего периферийного блока, разблокирует задачу «Задача 2» являющуюся задачей, осуществляющей отложенную обработку прерывания и прекращает работу. В момент времени «B» управление передается «задаче 2», и она осуществляет обработку данных полученных от периферийного блока. После того, как алгоритм «задачи 2» завершит обработку, она вновь перейдет в состояние блокировки, что даст возможность «Задаче 1» продолжить свою работу.

Существует масса сценариев, различающихся соотношением приоритетов, присвоенных задачам 1 и 2 приведенного выше примера. Что действительно важно, так это найти такие условия, когда большинство кода будет выполняться в контексте операционной системы с использованием всех сервисов, предоставляемых ее ядром и под контролем запущенного планировщика при сохранении приемлемых (допустимых) временных задержек реакции на события, происходящие в реальном времени.

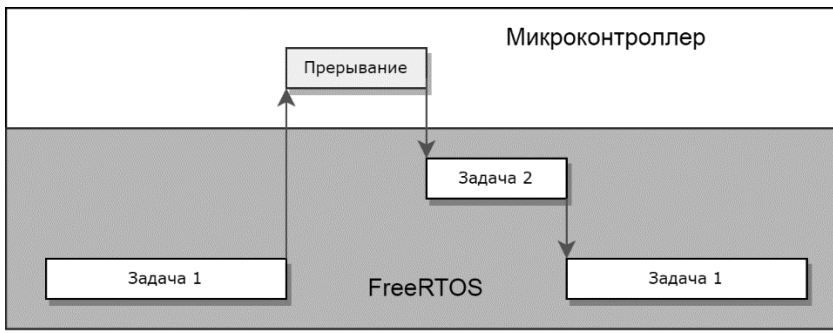


Рисунок 41. Контекст выполнения.

Не существует каких-либо правил или требований относительно того, в каких ситуациях лучше выполнять всю обработку, вызванную прерыванием в обработчике, а в каких ситуациях лучше отложить часть обработки, перенеся ее в задачу. Мы лишь можем определить некоторые ключевые моменты, являющиеся важными для принятия решения:

- обработка, вызванная прерыванием, нетривиальна. Например, если прерывание просто сохраняет результат аналого-цифрового преобразования, это лучше всего выполнить внутри

ISR, но если результат преобразования должен быть пропущен через программный фильтр, то логично перенести эти действия в задачу.

- Необходимо выполнить какие-либо действия сложно совместимые с обработкой прерывания. Например, отправить информацию по интерфейсу UART.
- Код, который нужно поместить в обработчик прерывания, не является детерминированным — это означает, что заранее неизвестно, сколько времени займет выполнение данного кода.

Стоит отметить и тот факт, что перенос определенного объема кода из обработчика прерывания в задачу положительно скажется на учете времени ядром операционной системы и не приведет к появлению сбоев в работе блокировок задач по времени.

Как вы наверняка догадались, весь механизм реализации идеи отложенной обработки прерывания реализуется за счет синхронизации обработчика прерывания с задачей. Для синхронизации можно использовать несколько инструментов предоставляемых ядром ОС. С одним из них мы уже успели познакомиться. Это очереди.

Бинарный семафор

Давайте вернемся к рисунку 40. Если мы пытаемся реализовать концепцию отложенной обработки прерывания, то нам необходим инструмент, который после окончания работы обработчика прерывания передал бы управление коду «Задача 2» и обеспечил бы полный цикл выполнения кода задачи.

Передачу управления в «Задачу 2» мы легко сможем сделать за счет присвоения данной задаче высокого приоритета, это гарантирует, что в определенный момент она окажется единственной задачей с самым большим приоритетом. В дополнение мы можем включить вызов макроса portYIELD_FROM_ISR() в код обработчика прерывания, что гарантирует смену контекста сразу после выхода из обработчика.

Остается только решить, какой инструмент использовать для блокировки «Задачи 2».

Инструменты, описываемые в этой и некоторых последующих главах, будут использовать блокировку задач, как способ добиться их взаимной синхронизации. Давайте представим простейшую программу, описанную в Главе 1. Это температурный контроллер, каждая задача которого выполняет строго определенную функцию: производит измерение температуры, выполняет фильтрацию данных, отображает значение на экране. Для того, чтобы этот комплекс обособленных задач мог функционировать совместно в рамках единого программного обеспечения мы нуждаемся в механизме синхронизации. Один такой инструмент – очереди, нами уже изучен. И действительно очередь способна блокировать задачи как мы это и рассмотрели на практических примерах.

Теперь давайте представим себе очередь длиной в 1 элемент. Очередь будет иметь только два состояния «очередь полна» или «очередь пуста». По сути, она становится бинарной, имеет только два состояния. А значит, задача, осуществляющая попытку чтения из очереди длиной в 1 элемент или будет выполнена, или перейдет в задачу блокировки, ожидая появление элемента. Это и является иллюстрацией того, как работают семафоры.

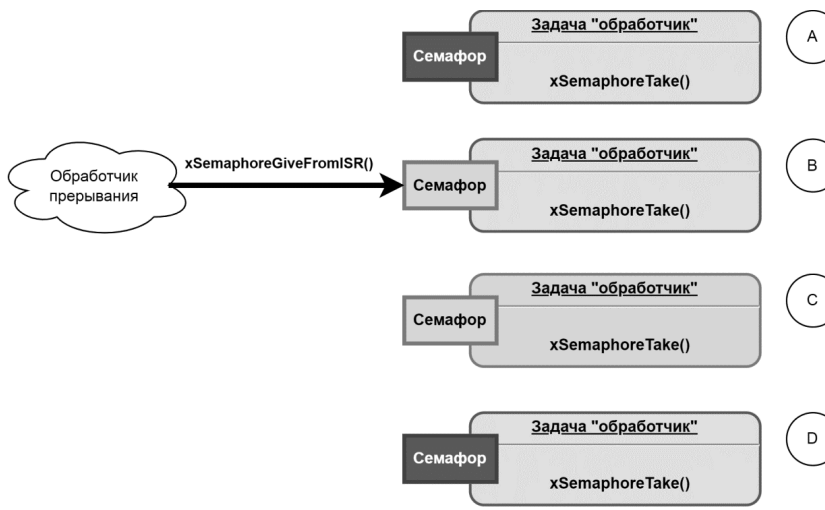


Рисунок 42. Использование бинарных семафоров для синхронизации.

Рисунок 42 показывает, как мы можем использовать бинарные семафоры для синхронизации задачи и обработчика прерывания. (A) – семафора нет, при попытке выполнить вызов API функции xSemaphoreTake() задача – обработчик отложенного прерывания переходит в состояние блокировки. (B) – обработчик прерывания выполнив свою часть работы вызывает функцию xSemaphoreGiveFromISR() и предоставляет семафор. (C) – задача – обработчик считывает семафор обнуляя его и начинает выполнение своего цикла. (D) – семафора нет, при попытке выполнить вызов API функции xSemaphoreTake() задача – обработчик отложенного прерывания переходит в состояние блокировки.

Создание бинарного семафора

До того, как семафор будет впервые использован, его необходимо создать. Для создания бинарного семафора используется функция xSemaphoreCreateBinary().

```
SemaphoreHandle_t xSemaphoreCreateBinary( void );
```

Возвращаемое значение Возвращается NULL - семафор не может быть создан, так как для FreeRTOS недостаточно памяти в куче. Значение, отличное от NULL, указывает на то, что семафор был успешно создан. Необходимо сохранить как дескриптор созданного семафора.

«Взять» семафор xSemaphoreTake()

«Взять» или «получить» семафор. Семафор можно взять, только если он был создан. Эта функция может быть использована для всех типов семафоров, кроме рекурсивных мьютексов.

Функцию xSemaphoreTake() нельзя использовать в обработчике прерывания.

```
BaseType_t xSemaphoreTake( SemaphoreHandle_t xSemaphore,
                           TickType_t xTicksToWait );
```

xSemaphore Семафор, который мы собираемся «взять». На семафор ссылается переменная типа

SemaphoreHandle_t.

xTicksToWait Максимальное время, в течение которого задача, ожидающая семафора должна оставаться в заблокированном состоянии. Таймаут.

Возвращаемое значение Возможны два возвращаемых значения:

1. pdPASS – семафор успешно получен.

2. pdFALSE - Семафор недоступен. Если был использован таймаут (xTicksToWait не равно нулю), то вызывающая задача будет помещена в состояние блокировки в ожидании доступности семафора. Данное значение указывает на тот факт, что время ожидания истекло.

«ДАТЬ» семафор xSemaphoreGiveFromISR()

Как двоичные, так и счетные семафоры можно «дать» с помощью функции xSemaphoreGiveFromISR(). По сути это версия функции xSemaphoreGive(), безопасная для использования внутри обработчика прерывания.

В начале главы мы разбирали функцию параметра pxHigherPriorityTaskWoken, вызов функции xSemaphoreGiveFromISR() отличается то xSemaphoreGive() только наличием этого параметра.

```
BaseType_t xSemaphoreGiveFromISR( SemaphoreHandle_t xSemaphore,
                                  BaseType_t *pxHigherPriorityTaskWoken );
```

xSemaphore Семафор, который мы собираемся «дать».

pxHigherPriorityTaskWoken Возможна ситуация, когда один семафор будет являться причиной блокировки нескольких задач.

Если вызов xSemaphoreGiveFromISR() приведет к тому, что выдаваемый семафор послужит причиной разблокировки задачи с приоритетом выше, чем у задачи

выполнявшейся до входа в обработчик прерывания для *pxHigherPriorityTaskWoken будет установлено значение pdTRUE.

Если вызов xSemaphoreGiveFromISR() устанавливает это значение в pdTRUE, то перед выходом из прерывания должно выполняться переключение контекста.

Возвращаемое значение

Возможны два возвращаемых значения:

1. pdPASS – успешный вызов функции.

2. pdFALSE – семафор уже выдан и не может быть выдан повторно.

Синхронизация прерывания и задачи

Давайте на практике рассмотрим работу двоичных семафоров как средства синхронизации обработчиков прерывания и задач.

Для начала сконфигурируем таймер для формирования периодических прерываний. Мы будем использовать таймер TIM3. Зададим для таймера временной интервал длительностью в 2,5 миллисекунды. Эта частота отличается от частоты тиковых прерываний.

```
static void MX_TIM3_Init(void)
{
  TIM_ClockConfigTypeDef sClockSourceConfig = {0};
  TIM_MasterConfigTypeDef sMasterConfig = {0};

  htim3.Instance = TIM3;
  htim3.Init.Prescaler = 720;
  htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim3.Init.Period = 250;
  htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim3.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_Base_Init(&htim3) != HAL_OK)
  {
    Error_Handler();
  }
  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  if (HAL_TIM_ConfigClockSource(&htim3, &sClockSourceConfig) != HAL_OK)
```

```
  {
    Error_Handler();
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim3, &sMasterConfig) != HAL_OK)
  {
    Error_Handler();
  }
}
```

Затем напишем код, выполняемый в обработчике прерывания от таймера. Для отображения момента возникновения прерывания мы используем 3-й порт ввода-вывода. Будем изменять его состояние.

```
void TIM3_IRQHandler(void)
{
  HAL_TIM_IRQHandler(&htim3);

  BaseType_t xHigherPriorityTaskWoken;

  GPIOA->BSRR = GPIO_PIN_3;

  xHigherPriorityTaskWoken = pdFALSE;
  xSemaphoreGiveFromISR(myBinarySem01Handle, &xHigherPriorityTaskWoken);

  GPIOA->BSRR = (uint32_t)GPIO_PIN_3 << 16U;
  portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

Задачи 1 и 2 оставим в том, же виде, что и в большинстве предыдущих примеров. Они имеют нормальный приоритет и будут занимать все доступное время, разделяя его поровну.

```
void StartTask01(void const * argument)
{
  for(;;)
  {
    __NOP();
  }
}

void StartTask02(void const * argument)
{
  for(;;)
  {
    __NOP();
```

```
  }
}
```

Задача 3 будет задачей с приоритетом выше нормального и будет находиться в блокировке ожидая появления семафора. Ее код приведен ниже.

```
void StartTask03(void const * argument)
{
  for(;;)
  {
    xSemaphoreTake( myBinarySem01Handle, portMAX_DELAY);

    // User code
    GPIOA->BSRR = GPIO_PIN_2;
    for(uint32_t x=0; x<8000; x++) __NOP();
    GPIOA->BSRR = (uint32_t)GPIO_PIN_2 << 16U;
  }
  /* USER CODE END StartTask03 */
}
```

Для имитации полезного кода в задаче определен оператор цикла.

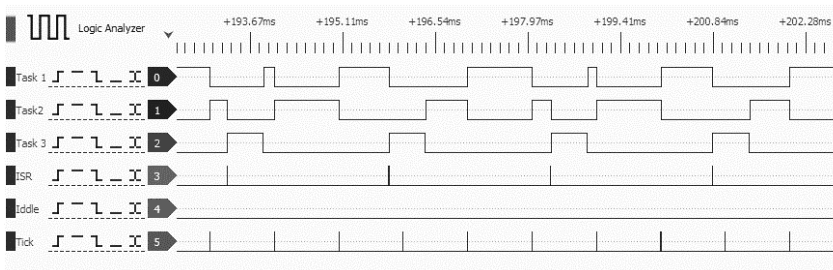


Рисунок 43. Синхронизация прерывания и задачи.

Обратите внимание на приведенный выше график. Мы можем видеть, что idleTask не вызывается. Все доступное время планировщик поровну распределяет между задачами 1 и 2. Каждые 2,5 миллисекунды возникает прерывание от таймера. В обработчике прерывания происходит «выдача» семафора. После этого код задачи 3 определяет, произошла ли разблокировка высоко приоритетной задачи, и, если это имеет место, то указывает планировщику на необходимость переключения контекста.

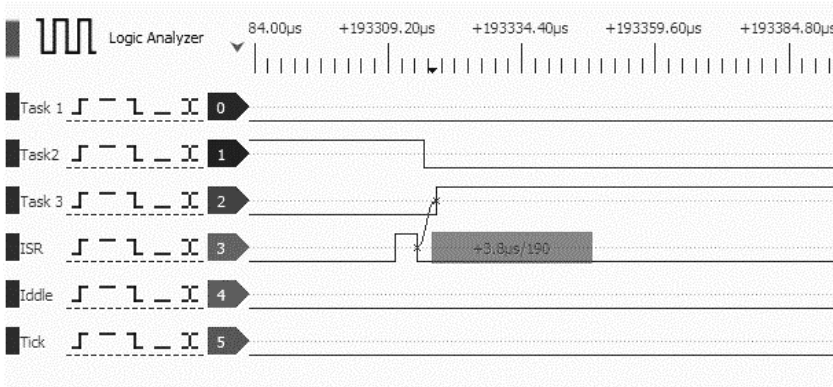


Рисунок 44. Время передачи управления от обработчика прерывания к задаче.

Обратите внимание на то, что от завершения кода, выполняемого внутри обработчика прерывания, и до начала выполнения кода задачи 3 проходит около 3,8 микросекунды. Это достаточно хороший результат.

Счетный семафор

Трудно отрицать тот факт, что семафоры являются крайне удобным и быстрым инструментом синхронизации. В начале предыдущего раздела мы рассматривали семафоры как очереди длиной в один элемент. По сути, если вы заглянете в исходный код FreeRTOS или ее портов, например, CMSIS-RTOS, то обнаружите именно очередь. Вместе с тем, уменьшая количество элементов очереди до одного, мы все-равно оказываемся в выигрыше. Мы экономим место в оперативной памяти, мы не занимаемся индексированием нескольких элементов очереди. Однако, оставим это, вернемся к семафорам.

Что если объединить идеи очередей и семафоров? Например, сделать очередь длиной больше одного элемента, но абсолютно игнорировать любые данные в этой очереди обращая внимание только на количество элементов. Это и будет счетным семафором.

Чаще всего, счетные семафоры используются для двух целей:

- Подсчет событий. В этом сценарии некий обработчик событий будет «отдавать» семафор каждый раз, когда происходит

событие. Это будет приводить к тому, что значение счетчика семафора при каждом событии будет увеличиваться. Каждый раз, когда задача обрабатывающая событие будет «брать» семафор, значение счетчика будет уменьшаться. Таким образом, значение счетчика - это разность между количеством произошедших событий и количеством обработанных событий. Часто счетные семафоры, используемые для подсчета событий, создаются с начальным значением счетчика.

- Управление ресурсами. В этом сценарии значение счетчика указывает количество доступных ресурсов. Чтобы получить контроль над ресурсом, задача должна сначала получить семафор, уменьшив значение счетчика семафора. Когда значение счетчика достигает нуля, свободных ресурсов нет. Когда задача завершает работу с ресурсом, она «возвращает» семафор, увеличивая значение счетчика семафора.

Выглядит несколько формально, не правда ли? На самом деле у этого есть и практическое применение. Например, при реализации некоторого станка мы вполне можем создавать некоторую задачу, реализующую элементарное действие. К примеру, задачу, реализующую единичное перемещение какого-либо элемента станка. В этом случае счетный семафор может стать прекрасным инструментом управления такой задачей. Задав некоторое значение счетного семафора мы тем самым определим количество элементарных действий для задачи, реализующей эти действия.

Для того, чтобы использование функционала счетных семафоров стало возможным, необходимо в файле конфигурации FreeRTOSConfig.h определить константу configUSE_COUNTING_SEMAPHORES и присвоить ей значение 1.

Создание счетного семафора

Прежде чем использовать счетный семафор (как и любой элемент ядра операционной системы) его необходимо создать вызовом функции API - xSemaphoreCreateCounting().

```
SemaphoreHandle_t xSemaphoreCreateCounting( UBaseType_t uxMaxCount,
                                            UBaseType_t uxInitialCount );
```

uxMaxCount	Максимальное значение, до которого семафор может осуществлять счет. Продолжая аналогию с очередью, значение uxMaxCount равно длине очереди.
uxInitialCount	Начальное значение счетчика, которое будет установлено в момент создания семафора.
Возвращаемое значение	NULL – если семафор не может быть создан по причине недостаточности памяти в куче. Значение отличное от нуля – семафор создан. Это значение является дескриптором семафора.

Практический пример

Возьмем за основу исходный код созданный для демонстрации бинарных семафоров и немного модернизируем его. Для этого изменим код обработчика прерывания. Теперь он будет добавлять в счетный семафор не один, а целых три токена.

```
void TIM3_IRQHandler(void)
{
  HAL_TIM_IRQHandler(&htim3);

  BaseType_t xHigherPriorityTaskWoken;

  GPIOA->BSRR = GPIO_PIN_3;

  xHigherPriorityTaskWoken = pdFALSE;

  xSemaphoreGiveFromISR(myCountingSem01Handle, &xHigherPriorityTaskWoken);
  xSemaphoreGiveFromISR(myCountingSem01Handle, &xHigherPriorityTaskWoken);
  xSemaphoreGiveFromISR(myCountingSem01Handle, &xHigherPriorityTaskWoken);

  GPIOA->BSRR = (uint32_t)GPIO_PIN_3 << 16U;

  portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

Код задачи 3, имеющей приоритет выше нормального и имитирующей отложенный обработчик прерывания мы тоже немного

модифицируем, добавим в него один вызов ассемблерной инструкции nop; - для создания минимально возможной задержки между выполнением итераций цикла задачи 3.

```
void StartTask03(void const * argument)
{
  for(;;)
  {
    xSemaphoreTake( myCountingSem01Handle, portMAX_DELAY);
    // User code
    GPIOA->BSRR = GPIO_PIN_2;
    for(uint32_t x=0; x<1000; x++) __NOP();
    GPIOA->BSRR = (uint32_t)GPIO_PIN_2 << 16U;
    __NOP();
  }
}
```

Как можно предположить, после каждого выполнения обработчика прерывания должна происходить разблокировка Задачи 3, имеющей приоритет выше нормального. И так как в обработчике прерывания в счетный семафор отправляются целых три токена, Задача 3, переходя в состояние готовности, тоже должна выполнить три итерации цикла.

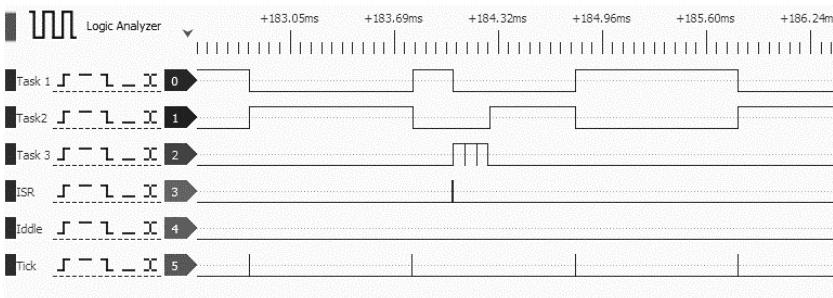


Рисунок 45. Счетный семафор.

Приведенная диаграмма полностью подтверждает данные ожидания. Мы видим, что за вызовом функции обработчика прерывания следуют три цикла выполнения задачи 3.

Эффективность дизайна

В предыдущей главе мы рассматривали интересный пример, касающийся взаимной блокировки задач при использовании очередей. Был приведен график (рисунок 46) иллюстрирующий последовательность выполнения задач. Немного детализируем этот рисунок.

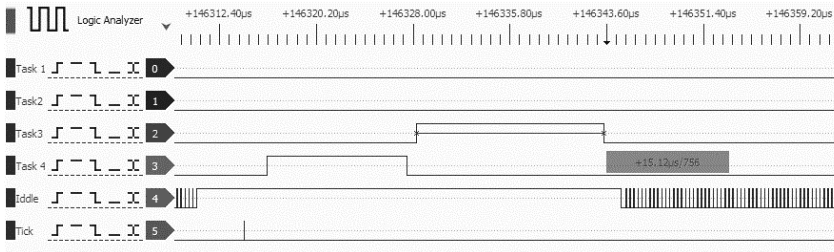


Рисунок 46. Блокировка задач очередью.

Напомню, в показанном примере задача 4 читает данные из очереди, а задача 3, отправляет данные в очередь. В определенный момент, очередь оказалась переполнена и теперь, задача 3 может отправить данные только после того, как освободится свободное место в очереди. Этим и объясняется последовательность выполнения задач 3, 4.

Как видно из диаграммы, на отправку данных в очередь было потрачено 15,2 микросекунды и еще 800 наносекунд на переключение от одной задачи к другой. Это хороший повод задуматься о эффективности.

На примере главы, посвященной задачам, мы знаем, что переключение задач занимает примерно 600 наносекунд. Плюс 1,8 микросекунды на работу самого планировщика, в итоге на переключение между задачами будет потрачено 0,24% доступного времени. Разумеется, это весьма приблизительная оценка, не учитывающая множество факторов.

Теперь представим, что мы приняли решение использовать механизм отложенной обработки прерывания для обслуживания высокоскоростного UART. Также мы планируем использовать очередь для отправки данных из обработчика прерывания в задачу. Как видно из рисунка 46 функция, помещающая информацию в очередь, тратит на это 15,2 микросекунды. Для приема информации на скорости 57600 бод

только на обслуживание очереди будет потрачено 88% всего доступного времени. Работа на больших скоростях при таком дизайне будет невозможна. Вы только задумайтесь, 88% времени на обслуживание функционала операционной системы.

Приведенный пример вовсе не является свидетельством того, что операционная система работает неправильно или мы должны отказаться от ее использования. Это свидетельствует только о то, что мы выбрали неверный вариант дизайна приложения. Ведь очередь является весьма неэффективным инструментом, если данные поступают с высокой частотой.

Если нам требуется высокая производительность, то очевидно мы должны использовать немного другие приемы:

- Использовать механизмы прямого доступа к памяти (DMA) для операций приема и буферизации поступающей информации. Затем мы можем использовать либо семафор, либо уведомление для того, чтобы разблокировать задачу, обрабатывающую полученную в буфер информацию.
- В теле обработчика прерывания копировать полученные данные в потокобезопасный буфер в ОЗУ. Далее использовать нотификацию для оповещения обрабатывающей задачи о готовности данных.
- Оставить обработку в обработчике прерывания с последующим использованием механизма очередей для последующей отправки в задачу ссылки на буфер содержащий информацию.

Хочется отдельно подчеркнуть, что проверка дизайна приложения использующего операционную систему FreeRTOS на правильность и эффективность является очень важным этапом разработки и будет подробно рассмотрено в последующих главах.

Вложенность прерываний

Говоря о возможностях, предоставляемых операционной системой мы используем слово «приоритет» и в отношении задач, и в отношении приоритета прерываний. Из-за этого часто возникает путаница. В данном разделе мы будем обсуждать вопросы, связанные с приоритетами

прерываний. То есть приоритеты, с которыми выполняются функции обработчики прерываний (ISR) по отношению друг к другу.

Необходимо помнить, что приоритет, назначенный задаче, никоим образом не связан с приоритетом, назначенным прерыванию. Аппаратное обеспечение микроконтроллера решает, когда будет выполняться обработчик прерывания (ISR), тогда как программное обеспечение (планировщик) решает, когда будет выполняться задача.

Обработчик прерывания (ISR), выполняемый в ответ на аппаратное прерывание, прерывает задачу, но задача не может прервать обработчик.

Для правильного функционирования операционной системы FreeRTOS на архитектурах, поддерживающих приоритеты и вложенность прерываний в конфигурационном файле FreeRTOSConfig.h должны быть определены некоторые константы.

configMAX_SYSCALL_INTERRUPT_PRIORITY или configMAX_API_CALL_INTERRUPT_PRIORITY	- устанавливает наивысший приоритет прерывания, из которого могут быть вызваны безопасные для использования внутри обработчиков прерываний функции FreeRTOS API. Функции, оканчивающиеся на «FromISR».
configKERNEL_INTERRUPT_PRIORITY	Устанавливает приоритет прерывания, используемый тактовым прерыванием. Необходимо чтобы был установлен на самый низкий возможный приоритет прерывания. Если порт FreeRTOS под вашу архитектуру не использует константу configMAX_SYSCALL_INTERRUPT_PRIORITY, то любое прерывание, использующее безопасные функции API FreeRTOS, также должно выполняться с приоритетом, определенным параметром configKERNEL_INTERRUPT_PRIORITY.

Глава 7. Программные таймеры

Начиная работу с микроконтроллерами каждый программист достаточно быстро убеждается в удобстве использования аппаратных таймеров. Это надежный и очень гибкий инструмент, позволяющий нам получить отметки времени, измерить временные интервалы, получить импульсы и прерывания с необходимой частотой.

Но есть и очевидные недостатки:

- Аппаратный таймер использует механизм прерываний. Это достоинство, когда нам важен временной интервал, например, при организации динамической индикации и подобных процессах, однако в некоторых случаях использование системы обработки прерываний может стать нежелательным из-за связанной с ней необходимостью прерывания выполнения высоко приоритетной задачи.
- Аппаратных таймеров может не хватать в микроконтроллерах младших моделей.

Операционная система FreeRTOS предоставляет нам функционал программных таймеров, использующихся для планирования выполнения функции в установленное время в будущем или периодически с фиксированной частотой. Функция, выполняемая программным таймером, называется функцией обратного вызова (Callback) программного таймера.

Программные таймеры реализованы ядром FreeRTOS и находятся под его управлением. Они не требуют аппаратной поддержки и не связаны с аппаратными таймерами или аппаратными счетчиками. У текущей реализации аппаратных таймеров в операционной системе FreeRTOS есть очевидное преимущество – если таймер не выполняется и не происходит вызов функции обратного вызова, то такой программный таймер не использует процессорное время.

Функционал программных таймеров не является обязательным, это всего лишь расширение функционала. Для того, чтобы активировать программные таймеры необходимо:

1. Включить файл FreeRTOS/Source/timers.c в состав создаваемого вами проекта.
2. Установите для константы configUSE_TIMERS определенной в конфигурационном файле FreeRTOSConfig.h значение равное 1.

Как и задачи функции обратного вызова программного таймера реализованы как функции Си. Единственное, что в них особенного, это их прототип, который должен возвращать void и принимать дескриптор программного таймера в качестве единственного параметра.

```
void ATimerCallback( TimerHandle_t xTimer );
```

При срабатывании функция обратного вызова программного таймера выполняются от начала до конца и завершаются обычным образом. Из этого следует, что она должна быть короткой (не занимать процессорное время) и ни в коем случае не переходить в состояние «Заблокировано».

На самом деле, прочитав уже несколько абзацев этой главы можно догадаться как именно реализован функционал программных таймеров в операционной системе FreeRTOS. Функции обратного вызова программного таймера выполняются в контексте задачи, которая создается автоматически при запуске планировщика FreeRTOS. Именно по этой причине очень важно, чтобы функции обратного вызова программного таймера никогда не вызывали функции API FreeRTOS, которые приведут к тому, что вызывающая задача перейдет в состояние «Заблокировано». Можно вызывать такие функции, как xQueueReceive(), но только в том случае, если параметр функции xTicksToWait (который указывает время таймаута) установлен в 0. Недопустимо вызывать такие функции, как vTaskDelay().

Для каждого программного таймера нам необходимо определить «Период» — это время между запуском программного таймера и выполнением функции обратного вызова программного таймера.

По логике работы таймер может быть настроен как таймер однократного запуска и периодический. Таймер однократного запуска выполнит свою функцию обратного вызова только один раз. Таймер однократного запуска можно перезапустить вручную, но он не перезапустится сам. Периодический таймер после запуска будет

перезапускаться каждый раз по истечении заданного периода, что приводит к периодическому выполнению его функции обратного вызова.

В связи с тем, что программный таймер выполняется в контексте задачи FreeRTOS у него, также существуют определенные состояния:

- «Бездействие» - программный таймер существует, и на него можно сослаться по его дескриптору, но он не запущен, поэтому его функция обратного вызова не будет выполняться.
- «Работает» - программный таймер выполнит свою функцию обратного вызова по истечении времени, равного его периоду, с момента перехода программного таймера в состояние «Работает» или с момента последнего сброса программного таймера.

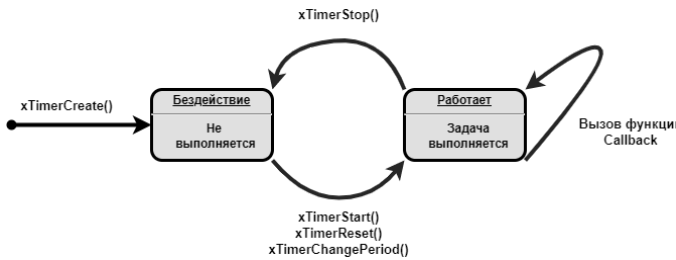


Рисунок 47. Алгоритм работы программного таймера работающего в периодическим режиме.

На рисунке 47 показан алгоритм работы программного таймера, запущенного в периодическом режиме.

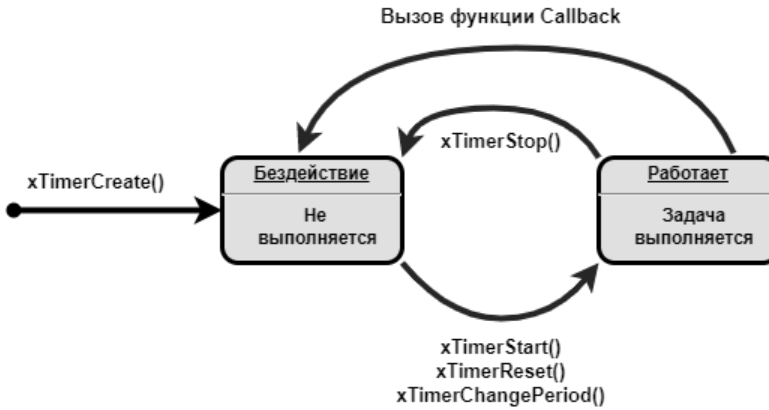


Рисунок 48. Алгоритм работы программного таймера в режиме однократного запуска.

На рисунке 48 показан алгоритм работы программного таймера запущенного в режиме однократного запуска.

Как видно из рисунков, представленных выше, между таймером, запущенным в режиме однократного запуска и таймером работающем в периодическом режиме разница только в том, в каком режиме окажется таймер после выполнения вызова функции обратного вызова.

Функция API xTimerDelete() удаляет таймер. Таймер можно удалить в любой момент.

Контекст программного таймера

Если вы активируете поддержку программных таймеров, определив константу configUSE_TIMERS в конфигурационном файле, то при запуске планировщика задач будет создана задача, которая возьмет на себя функцию службы таймера - задача демона RTOS. Все функции обратного вызова программного таймера будут выполняться в контексте одного и того же демона RTOS (или «службы таймера»).

В более ранних версиях FreeRTOS эта задача называлась «timer service task» потому, что первоначально это была выделенная задача только для реализации функционала программных таймеров. В более поздних версиях операционной системы эта задача стала более

универсальной и используется в том числе и для других целей. Теперь она известна под более общим названием «RTOS daemon task».

Для задачи демона, также, как и для всех остальных задач, можно определить приоритет и размер стека. Эти параметры устанавливаются константами configTIMER_TASK_PRIORITY и configTIMER_TASK_STACK_DEPTH. Обе константы определены в файле конфигурации FreeRTOSConfig.h.

Пожалуйста, обратите внимание, что для планировщика FreeRTOS задача демона ничем не отличается от остальных других задач FreeRTOS. Он будет обрабатывать команды или выполнять функции обратного вызова таймера только тогда, когда это позволит приоритет «RTOS daemon task» и политика планирования.

Очередь команд таймера

Порой бывает достаточно любопытным взглянуть на то, как API операционной системы функционирует, что называется «под капотом». Вполне логично опираться на уже имеющийся функционал, а не изобретать что-то новое, тем самым не плодить новый код и не создавать новые функции. К примеру, парой абзацев выше речь шла о том, что все существование программных таймеров - это не более чем специализированная задача. В свою очередь взаимодействие с этой задачей построено на использовании стандартного функционала очередей. Таким образом, функции API программного таймера работают за счет отправки команд из вызывающей задачи задаче демона в очередь, называемую «очередью команд таймера». Примеры команд включают «запустить таймер», «остановить таймер» и «сбросить таймер».

Очередь команд таймера — это стандартная очередь FreeRTOS, которая создается автоматически при запуске планировщика. Длина очереди команд таймера задается константой конфигурации времени компиляции configTIMER_QUEUE_LENGTH в FreeRTOSConfig.h.

Создание и запуск программного таймера

Для создания программного таймера может быть использована функция API xTimerCreate(). FreeRTOS V9.0.0 также включает функцию xTimerCreateStatic(), которая выделяет память, необходимую для статического создания таймера во время компиляции: Программный таймер должен быть явно создан, прежде чем его можно будет использовать.

На программные таймеры ссылаются переменные типа TimerHandle_t. xTimerCreate() используется для создания программного таймера и возвращает TimerHandle_t для ссылки на создаваемый программный таймер.

Программные таймеры создаются в состоянии бездействия. Их можно создавать до запуска планировщика или из задачи после запуска планировщика.

```
TimerHandle_t xTimerCreate( const char * const pcTimerName,
                            TickType_t xTimerPeriodInTicks,
                            UBaseType_t uxAutoReload,
                            void * pvTimerID,
                            TimerCallbackFunction_t pxCallbackFunction );
```

pcTimerName - описательное имя таймера.

xTimerPeriodInTicks - период таймера в тиках. Макрос pdMS_TO_TICKS() может использоваться для преобразования времени, указанного в миллисекундах, во время указывается в тиках.

uxAutoReload - параметр автоматической перезагрузки таймера. Установите uxAutoReload в значение pdTRUE, чтобы создать таймер с автоматической перезагрузкой. uxAutoReload в значении pdFALSE настраивает таймер в режиме однократного запуска.

pvTimerID - идентификатор таймера. ID является указателем и может использоваться разработчиком приложения для любых целей. Идентификатор особенно

полезен, когда одна и та же функция обратного вызова используется более чем одним программным таймером, поскольку его можно использовать для обеспечения памяти, специфичной для таймера.

pcCallbackFunction - функция обратного вызова программного таймера — это просто функция Си. Параметр pxCallbackFunction — это указатель на функцию (по сути, просто имя функции), которую следует использовать в качестве функции обратного вызова для создаваемого программного таймера.

Возвращаемое значение - в случае ошибки будет возвращен NULL, это возможно в случае, когда операционная система не может выделить необходимую память в куче. Значение, отличное от NULL, указывает на то, что программный таймер был успешно создан. Возвращаемое значение является дескриптором таймера.

Для запуска программного таймера находящегося в состоянии «Бездействия» используется функция xTimerStart(). Эта же функция может быть использована для сброса (перезапуска) программного таймера, находящегося в состоянии выполнения. Остановить программный таймер находящийся в состоянии выполнения – «Работает» можно вызвав функцию xTimerStop(). Остановка программного таймера аналогична переводу таймера в состояние бездействия.

Несмотря на то, что вызов функции xTimerStart() возможен и до фактического запуска планировщика задач, фактически отсчет времени и работа программного таймера начнутся только после того, как планировщик операционной системы будет фактически запущен.

```
BaseType_t xTimerStart( TimerHandle_t xTimer, TickType_t xTicksToWait );
```

xTimer - дескриптор программного таймера. Дескриптор создается функцией xTimerCreate() при создании программного таймера.

xTicksToWait - xTimerStart() отправляет в очередь команду, что позволяет указать задаче демону на необходимость запуска программного таймера. Параметр xTicksToWait определяет максимальное время таймаута при попытке отправки такого сообщения, если очередь уже заполнена. Таймаут указывается в тиках, можно использовать макрос pdMS_TO_TICKS() для преобразования времени, указанного в миллисекундах, во время, указанное в тиках.

Возвращаемое значение - pdPASS, если операция размещения команды на запуск программного таймера размещена в очереди успешно и pdFALSE, если разместить команду в очереди за отведённое время таймаута не удалось из-за переполнения очереди.

Рассмотрим процесс создания и запуска таймеров на простом примере:

```
#define mainONE_SHOT_TIMER_PERIOD pdMS_TO_TICKS( 3333 )
#define mainAUTO_RELOAD_TIMER_PERIOD pdMS_TO_TICKS( 500 )

int main( void )
{
  TimerHandle_t xAutoReloadTimer, xOneShotTimer;
  BaseType_t xTimer1Started, xTimer2Started;

  /* Создаем таймер однократного запуска xOneShotTimer. */
  xOneShotTimer = xTimerCreate("OneShot", mainONE_SHOT_TIMER_PERIOD,
                               pdFALSE, 0, prvOneShotTimerCallback );

  /* Создаем периодический таймер xAutoReloadTimer. */
  xAutoReloadTimer = xTimerCreate("AutoReload",
                                  mainAUTO_RELOAD_TIMER_PERIOD,
                                  pdTRUE, 0, prvAutoReloadTimerCallback );

  xTimer1Started = xTimerStart( xOneShotTimer, 0 );
  xTimer2Started = xTimerStart( xAutoReloadTimer, 0 );

  vTaskStartScheduler();

  for( ;; );
}
```

Соответственно для каждого созданного и запущенного программного таймера нам потребуется также создать функцию обратного вызова, которая и будет содержать выполняемый таймером код.

```
static void prvOneShotTimerCallback( TimerHandle_t xTimer )
{
  TickType_t xTimeNow;
  /* Получение текущего времени. */
  xTimeNow = xTaskGetTickCount();
  /* Вывод времени в консоль как подтверждение работы таймера. */
  vPrintStringAndNumber( "One-shot timer callback executing",
                         xTimeNow );

  ulCallCount++;
}
```

И для периодически выполняемого таймера

```
static void prvAutoReloadTimerCallback( TimerHandle_t xTimer )
{
  TickType_t xTimeNow;
  /* Получение текущего времени. */
  xTimeNow = uxTaskGetTickCount();
  /* Вывод времени в консоль как подтверждение работы таймера. */
  vPrintStringAndNumber( "Auto-reload timer callback executing",
                         xTimeNow );

  ulCallCount++;
}
```

Идентификатор таймера

Как было сказано выше, каждый программный таймер имеет идентификатор, представляющий собой значение тега, которое может использоваться разработчиком приложения для любых целей. Идентификатор хранится в пустом указателе (void *), поэтому может напрямую хранить целочисленное значение, указывать на любой другой объект или использоваться в качестве указателя на функцию.

Начальное значение присваивается идентификатору при создании программного таймера, после чего идентификатор можно обновить с

помощью функции API vTimerSetTimerID() и запросить с помощью функции API pvTimerGetTimerID().

В отличие от других функций API программного таймера, vTimerSetTimerID() и pvTimerGetTimerID() напрямую обращаются к программному таймеру — они не отправляют команду в очередь команд таймера.

Рассмотрим функции работы с идентификатором программного таймера немного подробнее.

```
void vTimerSetTimerID( const TimerHandle_t xTimer, void *pvNewID );
```

xTimer - дескриптор программного таймера. Дескриптор создается функцией xTimerCreate() при создании программного таймера.

pvNewID - новый идентификатор, который должен быть присвоен.

```
void *pvTimerGetTimerID( TimerHandle_t xTimer );
```

xTimer - дескриптор программного таймера. Дескриптор создается функцией xTimerCreate() при создании программного таймера.

Возвращаемое значение - идентификатор.

Попробуем воспользоваться идентификаторами программного таймера на простом примере. Мы создадим два программных таймера и будем использовать одну и ту же функцию обратного вызова для каждого из них. Наша функция обратного вызова должна будет самостоятельно определить в результате срабатывания какого таймера произошел ее вызов.

```
xOneShotTimer = xTimerCreate( "OneShot", mainONE_SHOT_TIMER_PERIOD,
```

```
                                pdFALSE, 0, prvTimerCallback );

xAutoReloadTimer = xTimerCreate( "AutoReload",
                                 mainAUTO_RELOAD_TIMER_PERIOD,
                                 pdTRUE, 0, prvTimerCallback );
```

Как видите, функция prvTimerCallback() будет выполняться по истечении времени любого таймера. Реализация prvTimerCallback() использует параметр функции, чтобы определить, была ли она вызвана из-за истечения времени одноразового таймера или из-за истечения времени таймера автоматической перезагрузки. prvTimerCallback() также демонстрирует, как использовать идентификатор программного таймера в качестве хранилища для конкретного таймера; каждый программный таймер ведет подсчет количества раз, когда он истек, в своем собственном идентификаторе, а таймер автоматической перезагрузки использует этот счет, чтобы остановить себя при пятом запуске.

```
static void prvTimerCallback( TimerHandle_t xTimer )
{
  TickType_t xTimeNow;
  uint32_t ulExecutionCount;

  ulExecutionCount = ( uint32_t ) pvTimerGetTimerID( xTimer );
  ulExecutionCount++;
  vTimerSetTimerID( xTimer, ( void * ) ulExecutionCount );

  xTimeNow = xTaskGetTickCount();

  if( xTimer == xOneShotTimer )
  {
     vPrintStringAndNumber( "One-shot timer callback executing",
                            xTimeNow );
  }
  else
  {
     vPrintStringAndNumber( "Auto-reload timer callback executing",
                            xTimeNow );
     if( ulExecutionCount == 5 )
     {
        xTimerStop( xTimer, 0 );
     }
  }
}
```

Изменение периода таймера

Описывая возможные состояния программных таймеров было отмечено, что такие действия, как сброс и изменение периода запускают программный таймер. Т.е. вместе с изменением отмеряемого таймером временного интервала происходит и его запуск.

Период программного таймера изменяется с помощью функции xTimerChangePeriod().

Если xTimerChangePeriod() используется для изменения периода уже запущенного таймера, то таймер будет использовать новое значение периода для пересчета времени его истечения. Пересчитанное время истечения относится к моменту вызова xTimerChangePeriod(), а не к моменту первоначального запуска таймера. Если xTimerChangePeriod() используется для изменения периода таймера, который находится в состоянии бездействия (таймер, который не работает), то таймер рассчитает время истечения и перейдет в состояние выполнения (таймер начнет работать) .

```
BaseType_t xTimerChangePeriod( TimerHandle_t xTimer,
                TickType_t xNewTimerPeriodInTicks,
                TickType_t xTicksToWait );
```

xTimer - дескриптор программного таймера. Дескриптор создается функцией xTimerCreate() при создании программного таймера.

xTimerPeriodInTicks - период программного таймера в тиках.

xTickToWait - таймаут. Период времени, в течение которого информация о смене периода и перезапуске программного таймера должна быть помещена в очередь.

Возвращаемое значение - pdPASS, если сообщение помещено в очередь программного таймера успешно и pdFALSE, если в течение отведенного таймаута поместить информацию в очередь управления программным

таймером не удалось.

Вместе с функцией изменения периода программного таймера при рассмотрении возможных состояний упоминалась и возможность сброса программного таймера, иными словами его повторного запуска; при этом время истечения таймера пересчитывается относительно того, когда таймер был сброшен, а не когда таймер был первоначально запущен.

Обратите внимание, что функция xTimerReset(), следуя логике организации программных таймеров, также может быть использована для запуска таймера, находящегося в состоянии Бездействия.

```
BaseType_t xTimerReset( TimerHandle_t xTimer, TickType_t xTicksToWait );
```

xTimer - дескриптор программного таймера. Дескриптор создается функцией xTimerCreate() при создании программного таймера.

xTickToWait - таймаут. Период времени, в течение которого информация о сбросе программного таймера должна быть помещена в очередь.

Возвращаемое значение - pdPASS, если сообщение помещено в очередь программного таймера успешно и pdFALSE, если в течение отведенного таймаута поместить информацию в очередь управления программным таймером не удалось.

Существует удачный пример демонстрирующий использование программного таймера и его основных функций. Представьте, что перед нами стоит задача управления подсветкой на сотовом телефоне. Подсветка должна включаться при нажатии на любую клавишу, при нажатии клавиш работа подсветки должна продляться и при отсутствии нажатий в течении определенного периода времени, подсветка должна автоматически отключаться.

Для реализации описанного выше алгоритма программный таймер подходит наилучшим образом. Достаточно организовать вызов функции xTimerReset() при каждом нажатии клавиши, а период таймера,

программируемый при его создании функцией xTimerCreate() установить равным желаемому времени работы подсветки.

Это хороший пример того, как использование встроенного функционала FreeRTOS эффективно сокращает количество кода, создаваемого разработчиком приложения, использующего данную операционную систему.

Практическое использование

Проведем несколько экспериментов и рассмотрим работу программного таймера в доступных режимах. Прежде всего опробуем таймер в режиме однократного запуска. Создадим функцию обратного вызова (Callback) для программного таймера.

```
void Callback01(void const * argument)
{
   GPIOA->BSRR = GPIO_PIN_3;
   // Действия таймера
   __NOP();
   GPIOA->BSRR = (uint32_t)GPIO_PIN_3 << 16U;
}
```

Для примера, единственное что будет делать наш программный таймер, это изменять состояние порта ввода-вывода. По этому изменению мы будем наблюдать за тем, как таймер работает.

Задачи 1 и 2 будут выполняться с нормальным приоритетом и их программный код ничем не будет отличаться от ранее рассмотренных примеров. Задача 3 будет инициализировать таймер и заставлять его отмерять интервал длительностью в 3 миллисекунды.

```
void StartTask03(void const * argument)
{
  xTimerChangePeriod(myTimer01Handle, 3, 10);
  for(;;)
  {
     xTimerStart(myTimer01Handle, 10);
     osDelay(10);
  }
}
```

Как видите, единственное, что делает данная задача, это инициализирует таймер, чтобы он отмерял период длительностью в 3 миллисекунды. Затем, задача будет запускать этот таймер и уходить в состояние блокировки на 10 миллисекунд. Таким образом, каждые 10 миллисекунд задача будет выполняться (периодическая задача) и запускать таймер, который должен сработать через 3 миллисекунды после задачи.

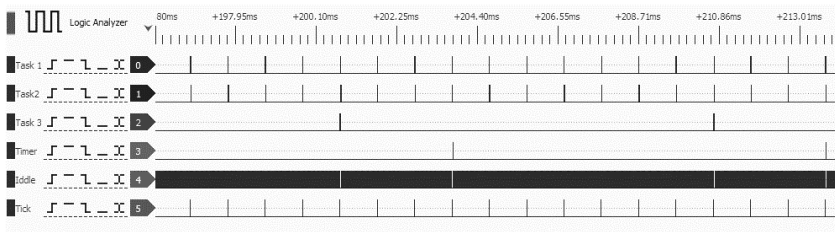


Рисунок 49. Программный таймер в режиме однократного запуска.

На рисунке 49 вы можете видеть, что таймер срабатывает, как это и ожидалось.

Теперь давайте опробуем таймер в периодическом режиме. Для этого при создании таймера функцией xTimerCreate(), параметр uxAutoReload определим равным pdTRUE. Задачу 3 изменим, теперь она будет обычной периодически выполняемой задачей, она будет запускать таймер один раз при своем запуске, дальнейшая работа таймера должна осуществляться периодически, независимо от задачи.

```
void StartTask03(void const * argument)
{
  xTimerChangePeriod(myTimer01Handle, 3, 10);
  for(;;)
  {
                osDelay(10);
  }
}
```

Как результат выполнения этого кода мы получим следующий график переключения задач:

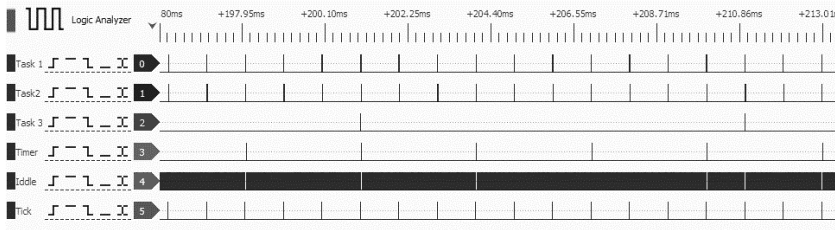


Рисунок 50. Программный таймер в периодическим режиме.

Обратите ваше внимание на то, что Задача 3 и функция обратного вызова программного таймера более не синхронизированы и выполняются независимо.

Обработка прерываний в задаче – демоне

В главе 6 мы рассмотрели концепцию отложенной обработки прерываний и механизм использования семафоров как средства синхронизации обработчика прерывания и задачи, на которую эта обработка отложена. Эта концепция подразумевает, что для каждого обработчика прерывания будет создана своя обособленная задача, которая займется отложенной обработкой.

При всех очевидных плюсах предложенной концепции есть и недостаток. Для каждой задачи выделяется и стек и формируется TCB (Task Control Block), занимающий некоторое место в памяти. Если же в создаваемом нами программном обеспечении используется механизм программных таймеров, то мы можем использовать уже имеющуюся задачу – демона, перенеся отложенную обработку прерываний в демона. Это можно реализовать за счет вызова API-функции xTimerPendFunctionCallFromISR(). Этот подход называется «централизованной отложенной обработкой прерывания».

Используя предложенные функции xTimerPendFunctionCall() и xTimerPendFunctionCallFromISR() мы отправляем задаче обслуживающей функционал программных таймеров команду выполнить ту или иную функцию. Эти функции будут выполняться в контексте FreeRTOS, под ее контролем и как следствие не окажут отрицательного влияния ни на контроль времени, ни на распределение задач в соответствии с их приоритетами.

Преимущества централизованной обработки отложенных прерываний включают:

- Низкое потребление ресурсов. При централизованной обработке мы избавлены от необходимости создавать отдельную задачу для каждого прерывания.
- Упрощенная модель. Функция обработчика – стандартная функция языка Си.

Есть у такого подхода и очевидные недостатки:

- Уменьшение гибкости. Вы не можете установить для каждого обработчика собственный приоритет. Каждая из используемых вами функций Си будет выполняться с приоритетом задачи демона. Приоритет задается константой configTIMER_TASK_PRIORITY, которая должна быть определена в конфигурационном файле FreeRTOSConfig.h.
- Меньше детерминизма. Вызов xTimerPendFunctionCallFromISR() отправляет команду в конец очереди команд таймера. Команды, которые уже находились в очереди команд таймера, будут обработаны задачей демона до того, как подойдет очередь команды «выполнить функцию».

С другой стороны, вы, как разработчик, понимаете, что у различных прерываний, у прерываний от разных источников требования к временным рамкам в течении которых это прерывание должно быть обработано – различаются. Это значит, что вы можете использовать оба предложенных метода отложенной обработки прерываний в сочетании с обработчиками, выполняющими свои действия вне операционной системы.

Централизация

Существуют две функции xTimerPendFunctionCallFromISR() и xTimerPendFunctionCall(), позволяющие отправить задаче демона RTOS информацию о необходимости отложенного выполнения какой-либо функции. Они имеют схожий набор параметров.

```
BaseType_t xTimerPendFunctionCallFromISR( PendedFunction_t xFunctionToPend,
```

```
                    void *pvParameter1, uint32_t ulParameter2,
                    BaseType_t *pxHigherPriorityTaskWoken );
```

Функция, выполнение которой поручается задаче – демону должна иметь следующий прототип:

```
void vPendableFunction( void *pvParameter1, uint32_t ulParameter2 );
```

Для вызова xTimerPendFunctionCallFromISR() определены следующие параметры:

xFunctionToPend - указатель на функцию, которая будет выполняться в задаче демона (по сути, просто имя функции). Прототип функции должен соответствовать показанному выше.

pvParameter1 - значение, которое будет передано в функцию, выполняемую задачей демона, в качестве параметра функции pvParameter1. Параметр имеет тип void *, что позволяет использовать его для передачи данных любого типа. Например, целые типы могут быть напрямую приведены к void *, альтернативно void * может использоваться для указания на структуру.

ulParameter2 - Значение, которое будет передано в функцию, выполняемую задачей демона, в качестве параметра функции ulParameter2.

pxHigherPriorityTaskWoken - xTimerPendFunctionCallFromISR() записывает в очередь команд таймера. Если задача демона находилась в заблокированном состоянии и ждала, пока данные станут доступными в очереди команд таймера, запись в очередь команд таймера приведет к тому, что задача демона выйдет из состояния блокировки. Если приоритет задачи демона выше приоритета выполняемой в данный момент

задачи (задачи, которая была прервана), то внутренне xTimerPendFunctionCallFromISR() установит для *pxHigherPriorityTaskWoken значение pdTRUE.
Если xTimerPendFunctionCallFromISR() устанавливает это значение в pdTRUE, то перед выходом из прерывания необходимо выполнить переключение контекста.

Возвращаемое значение - pdPASS - команда «выполнить функцию» была успешно передана в очередь команд таймера. pdFAIL - команда «выполнить функцию» не может быть помещена в очередь команд таймера, потому что очередь команд таймера уже заполнена.

Практическое использование демона

Централизованная отложенная обработка прерываний не будет вами использована в подавляющем большинстве проектов. Но, уверен, вы вспомните об этой возможности, когда потребуется использовать RTOS на микроконтроллерах с ограниченным количеством ресурсов. Опробуем перенос на задачу – демона на практическом примере.

Для иллюстрации сохраним Задачу 1 и Задачу 2, как имитацию нагрузки. Они будут выполнять минимум работы и демонстрировать нам тот факт, что планировщик переключает задачи и распределяет между ними доступное время. Настроим тайме TIM3 на генерацию прерываний. Прерывания будут генерироваться каждые 2,5 миллисекунды. Создадим обработчик прерываний от таймера TIM3.

```
void TIM3_IRQHandler(void)
{
  BaseType_t xHigherPriorityTaskWoken;
  xHigherPriorityTaskWoken = pdFALSE;

  HAL_TIM_IRQHandler(&htim3);

  GPIOA->BSRR = GPIO_PIN_2;
  __NOP();
```

```
  GPIOA->BSRR = (uint32_t)GPIO_PIN_2 << 16U;

  xTimerPendFunctionCallFromISR( vDeferredHandlingFunction, NULL, 0,
                                 &xHigherPriorityTaskWoken );
  portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

По изменению состояния порта ввода –вывода PA2 мы будем судить о состоянии обработчика прерываний. Как видно из кода, вызов функции xTimerPendFunctionCallFromISR() должен указать задаче демону на необходимость выполнения функции vDeferredHandlingFunction(). Как и сказано ранее, данная функция должна соответствовать прототипу.

```
static void vDeferredHandlingFunction( void *pvParameter1, uint32_t
ulParameter2 )
{
   GPIOA->BSRR = GPIO_PIN_3;
   for (uint32_t x = 0; x < 2000; x++) __NOP();
   GPIOA->BSRR = (uint32_t)GPIO_PIN_3 << 16U;
}
```

В нашем случае, внутри функции, мы будем только изменять состояние порта ввода-вывода PA3 и выполнять весьма небольшой «нагрузочный код».

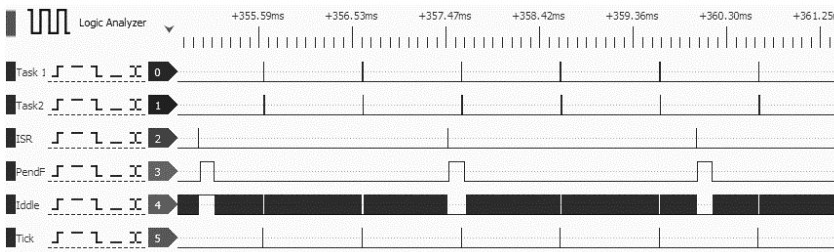


Рисунок 51. Обработка прерывания, отложенная на задачу – демона.

Мы можем достаточно наглядно наблюдать за тем, как сразу после выполнения обработчика прерывания происходит вызов функции vDeferredHandlingFunction, обозначенной на диаграмме как PendF.

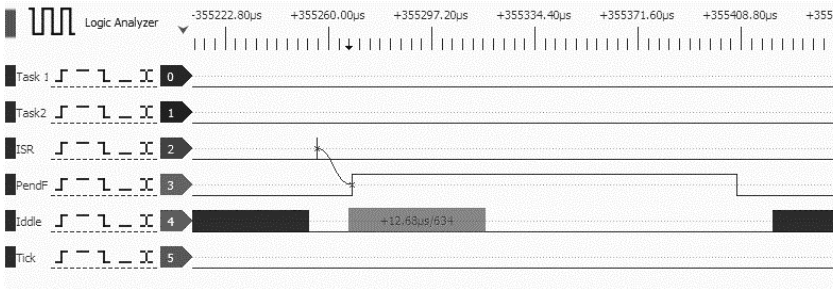


Рисунок 52. Обработка прерывания, отложенная на задачу – демона.

Между завершением работы обработчика прерывания и началом работы функции, выполняемой в рамках централизованной отложенной обработки прерывания, проходит 12,7 микросекунды.

Глава 8. Потокобезопасность

Потокобезопасность – свойство объекта или кода, которое гарантирует, что при исполнении или использовании несколькими потоками, код будет вести себя, как предполагается. Например, потокобезопасный счётчик не пропустит ни один счёт, даже если один и тот же экземпляр этого счётчика будет использоваться несколькими потоками.

Вся сложность заключается в том, что в многозадачной системе возможно возникновение ситуации, когда одна задача начинает обращаться к ресурсу, но в связи с тем, что работа с ресурсом занимает какое-то время, работа не завершается до того, как время, отведенное задаче, заканчивается. Если задача оставляет ресурс в несогласованном состоянии, то доступ к этому ресурсу любой другой задачи или прерывания может привести к повреждению данных или другой подобной проблеме. Рассмотрим на примерах:

1. Доступ к периферийным устройствам. Предположим, что в системе существуют две задачи. Каждая из задач пытается выполнить операцию вывода сообщения на дисплей. Задача «А» начинает выводить сообщение «**Температура сенсора**». В процессе вывода, по тиковому прерыванию происходит переключение с задачи «А» на задачу «Б». К моменту переключения задача «А» успела вывести только символы «**Темпера**». Задача «Б» приступает к выводу своей информации «**Подключение прервано**» и переходит в состояние блокировки. Задача «А» продолжает работу с того момента, когда она была прервана и в итоге на экран будет выведено сообщение «**ТемпераПодключение прерванотура сенсора**». Вы наверняка встречали такую ситуацию в некоторых устройствах.
2. Чтение, изменение, запись операций. В этой книге, для демонстрации процесса переключения между задачами используются порты ввода-вывода, переключаемые макросами операционной системы. В целях оптимизации и сокращения времени используется максимально быстро работающий код с прямым изменением регистра порта ввода-вывода:

```
GPIOA->BSRR = GPIO_PIN_1;
```

Однако, если мы посмотрим на ассемблерный код, получившийся в результате компиляции (использован Keil uVision), то увидим следующие строки:

```
MOVS            r1,#0x02
LDR             r2,[pc,#24]  ; @0x080012E0
STR             r1,[r2,#0x18]
```

Это не атомарная операция, для ее выполнения требуется более одной инструкции, и ее можно прервать. Если предположить, что существует сценарий, в котором две задачи пытаются обновить регистр с именем GPIOA, прерывая друг друга, то с большой долей вероятности содержимое регистра окажется непредсказуемым.

3. Неатомарный доступ к переменным. Обновление нескольких элементов структуры или изменение переменной, размер которой превышает естественный размер слова архитектуры, являются примерами неатомарных операций. Если они прерываются, они могут привести к потере или повреждению данных. Например, рассмотрим обновление значения 64 разрядной переменной:

```
uint64_t p64 = 0x0F0F0F0F0F0F0F0F;
```

ассемблерный код:

```
MOV             r1,#0xF0F0F0F
LDR             r2,[pc,#20]  ; @0x08001138
STR             r1,[r2,#0x00]
STR             r1,[r2,#0x04]
```

4. Реентерабельность функций. Функция является «реентерабельной», если данную функцию можно безопасно вызывать более чем из одной задачи одновременно. Т.е. функция построена таким образом, что данные не будут повреждены, а логические операции выполнены правильно.

Вы уже знакомы с тем, что каждая задача во FreeRTOS поддерживает свой собственный стек и собственный набор значений регистров процессора. Если функция не обращается ни

к каким данным, кроме данных, хранящихся в стеке или в регистре, то функция является реентерабельной и потокобезопасной. Пример такой функции вы видите ниже.

```
float Sum(const float first, const float second)
{
   float Sum;
   Sum = first + second;
   return Sum;
}
```

В качестве примера не реентерабельной функции можно привести функцию, результат которой зависит от внешних данных, которые могут быть изменены без какого-либо уведомления.

```
uint32_t var1;

uint64_t Inc100(void)
{
   uint64_t res;
   res = var1 + 100;
   return res;
}
```

Результат выполнения данной функции зависит от внешних данных.

Программистам с опытом хорошо известен метод «взаимного исключения». Этот метод применяется для согласования доступа к определенному ресурсу. Идея состоит в том, что как только задача или прерывание обратиться к общему ресурсу, она получит монопольный доступ и будет иметь его до тех пор, пока не прекратит доступ.

Изучаемая нами операционная система имеет несколько функций, которые можно использовать для реализации взаимного исключения, но лучший метод — это (по возможности) проектировать приложение таким образом, чтобы ресурсы не использовались совместно, а доступ к каждому ресурсу был только у одной задачи.

Критические секции кода

Выполнение кода создаваемой нами задачи может быть прервано в любой момент и заранее предсказать, когда задача продолжит свое выполнение порой достаточно сложно. В то же время существуют периферийные устройства, при работе с которыми мы должны придерживаться некоторых таймингов. Бывают и протоколы, требующие чтобы реакция или ответ, были сформированы в течении строго определенного временного промежутка.

Для решения этих задач операционная система предоставляет нам два макроса taskENTER_CRITICAL() и taskEXIT_CRITICAL(), определяющих критическую секцию кода выполнение которой не может быть прервано. Эти макросы не имеют каких-либо параметров или возвращаемых значений.

```
/* Начало критической секции кода */
taskENTER_CRITICAL();

GPIOA->BSRR |= 0x01;

/* Конец критической секции кода. */
taskEXIT_CRITICAL();
```

Необходимо признать, что описанный выше механизм критических секций кода представляет собой крайне грубый метод реализации взаимного исключения. Метод работает за счет отключения прерываний ниже уровня, установленного константой configMAX_SYSCALL_INTERRUPT_PRIORITY. Конкретная реализация макросов, реализующих критическую секцию кода, очень сильно зависит от возможностей используемой архитектуры микроконтроллера и кода, реализуемого портом.

Если использование критических секций кода является неизбежным, то постарайтесь, чтобы код был максимально коротким, иначе это отрицательно повлияет на время отклика на прерывания. Каждый вызов taskENTER_CRITICAL() должен быть связан с вызовом taskEXIT_CRITICAL(). Критические секции кода могут быть вложенными т.к. ядро операционной системы считает глубину вложений. Критическая секция будет закрыта только тогда, когда глубина вложенности вернется к нулю,

то есть когда один вызов taskEXIT_CRITICAL() был выполнен для каждого предыдущего вызова taskENTER_CRITICAL().

Существуют версии макросов taskENTER_CRITICAL() и taskEXIT_CRITICAL() позволяющие создавать критические секции кода и в обработчиках прерываний. taskENTER_CRITICAL_FROM_ISR() и taskEXIT_CRITICAL_FROM_ISR(). Эти макросы доступны только для портов FreeRTOS, которые реализуют вложенность прерываний.

Приостановка планировщика

Развивая вопросы реализации критических секций кода, безотносительно того, являются ли они хорошим решением или выбором «меньшего зла» из возможного, мы можем создать участки кода свободные от переключения контекста за счет временной приостановки функционирования планировщика. Приостановка планировщика иногда также называется «блокировкой» планировщика.

Существенная разница заключается в том, что критические секции, описанные разделом ранее, защищают область кода от доступа другими задачами и прерываниями. А критическая секция, реализованная путем приостановки планировщика, защищает только область кода от доступа других задач, поскольку прерывания остаются разрешенными. Об этой особенности нужно помнить и учитывать ее при планировании структуры создаваемого прикладного программного обеспечения.

Просто давайте посмотрим на это еще один раз. Только не с точки зрения фрагмента кода, который мы хотим выполнить, а с точки зрения следования основным принципам ОСРВ. И если для создаваемого ПО важно сохранить функционирование системы обработки прерываний, то правильным выбором будет именно приостановка планировщика.

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

Планировщик приостанавливается вызовом vTaskSuspendAll(). Приостановка планировщика прекращает переключение контекста, но оставляет прерывания включенными. Если прерывание запрашивает переключение контекста, в то время как планировщик приостановлен, то

запрос будет отложен, и выполнен только при возобновлении работы планировщика.

Логично, что и функции API FreeRTOS нельзя вызывать, пока работа планировщика приостановлена. Возобновить функционирование планировщика можно вызовом функции xTaskResumeAll().

```
BaseType_t xTaskResumeAll( void );
```

Вызовы vTaskSuspendAll() и xTaskResumeAll() могут быть вложенными. Ядро операционной системы производит подсчет глубины вложенности. Работа планировщика будет возобновлена только тогда, когда глубина вложенности вернется к нулю, то есть когда вызов xTaskResumeAll() был выполнен для каждого предыдущего вызова vTaskSuspendAll().

Глава 9. Снижение энергопотребления

Возможно, это одна из самых ожидаемых глав книги. В моем понимании специалисты эмбедеры делятся на две группы. Те, кто разобрался с методами снижения энергопотребления микроконтроллеров и те, кто мечтает найти подробное руководство.

Следует оговориться сразу. Не существует единого правильного подхода к снижению энергопотребления и повышению энергоэффективности. Это всегда сложный, комплексный подход, включающий как аппаратные решения (см. главу по энергопотреблению в «Микроконтроллеры. Практический курс»), так и программные алгоритмы работы с периферийными блоками и ядром микроконтроллера для достижения оптимальных значений потребления в различных режимах работы.

Самый простой и одновременно самый неэффективный метод - это воспользоваться механизмом idle task. Эта задача всегда существует в нашей системе, и мы легко можем инициировать функцию обратного вызова (Callback) которая будет вызываться каждый раз, когда планировщик не имеет других задач. С прототипом этой функции вы уже знакомы.

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

В этой функции вы можете реализовать переход микроконтроллера в режим с пониженным энергопотреблением, который будет прерван любым прерыванием. Например, прерыванием, извещающим планировщик о том, что очередной тик времени закончен.

```
void vApplicationIdleHook(void)
{
   __wfi();
}
```

Вполне рабочий подход, но он не позволит достичь большой экономии энергии. Прежде всего по той причине, что соотношение времени, затрачиваемого на вход-выход в состояние с пониженным

энергопотреблением ко времени проведенном в состоянии сна весьма невелико, а, следовательно, и не приносит большой экономии. Кроме того, такой упрощенный подход может использоваться только с режимами «неглубокого сна» (аналогично режиму Sleep в архитектуре ARM Cortex-M).

Между тем, планировщик FreeRTOS достаточно много знает о задачах, выполняемых под его контролем. Это дает возможность реализовать более эффективные алгоритмы. В частности, операционная система поддерживает режим «простоя без прерываний» - tickles idle.

В этом режиме планировщик FreeRTOS выявляет промежутки времени, когда какие-либо задачи, находящиеся в состоянии «Готова» отсутствуют. Если в результате планирования будет определено, что задачи заблокированы и в ближайшей перспективе не изменят своего состояния, то планировщик сможет инициировать переход в режим сна на более длительное время, а как следствие мы можем использовать более эффективные режимы энергопотребления. Для реализации режима простоя без прерываний операционная система использует макрос portSUPPRESS_TICKS_AND_SLEEP().

Макрос portSUPPRESS_TICKS_AND_SLEEP()

Для использования режима простоя без прерываний необходимо в конфигурационном файле FreeRTOSConfig.h определить константу configUSE_TICKLESS_IDLE. Возможны три варианта значения константы. Константа не определена или имеет значение «0» - TICKLESS IDLE не используется. Константа имеет значение «1» - используется встроенная реализация функции portSUPPRESS_TICKS_AND_SLEEP(), константа имеет значение «2» - будет использована пользовательская реализация portSUPPRESS_TICKS_AND_SLEEP().

Если режим TICKLESS IDLE активирован, ядро вызовет макрос portSUPPRESS_TICKS_AND_SLEEP(), если будут выполнены два условия:

1. Задача idle task - единственная задача, которая может выполняться, поскольку все остальные задачи либо заблокированы, либо приостановлены.
2. Планировщик планирует, что имеется не менее n полных квантов времени, прежде чем какая-либо задача выйдет из

состояния блокировки. n – задается константой configEXPECTED_IDLE_TIME_BEFORE_SLEEP, которая должна быть определена в конфигурационном файле FreeRTOSConfig.h.

Предложенный механизм реализации условий вхождения в portSUPPRESS_TICKS_AND_SLEEP() очень прост и логичен. Он позволяет не пытаться ввести микроконтроллер в режим пониженного энергопотребления при каждом вызове idle task, а дождаться ситуации, в которой планирование ожидает не менее n квантов времени подряд, что позволит избежать накладных расходов, связанных с потерей времени на вход-выход из режима пониженного энергопотребления.

TickLess Idle на практике

Определим константу configUSE_TICKLESS_IDLE присвоив ей значение 1 и посмотрим на работу режима tickles idle на практике:

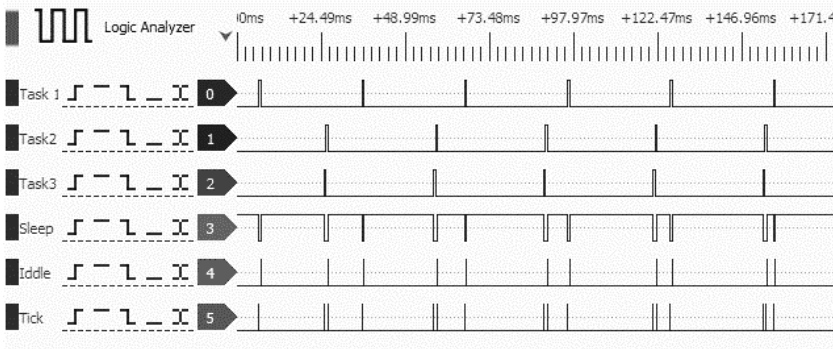


Рисунок 53. Режим tickles idle.

Внимательно посмотрите на представленную диаграмму. Задачи 1, 2 и 3 – периодические задачи, они имеют период 30, 31 и 32 миллисекунды соответственно. Выполняют свой код и переходят в состояние блокировки. График Sleep показывает время, в течении которого микроконтроллер находится в режиме с пониженным энергопотреблением. Idle – короткие моменты времени, когда планировщик вызывает задачу простоя. Tick – кванты времени.

В связи с тем, что все задачи – периодические, планировщик может с высокой точностью определить в течении какого времени переключение контекста не потребуется. На это время прекращается работа тиковых прерываний и микроконтроллер отправляется в режим с пониженным энергопотреблением.

Не всегда ситуация обстоит именно так. Как правило, наш код выполняется дольше чем показано на диаграмме выше, а паузы между задачами – короче. Исправим эту ситуацию.

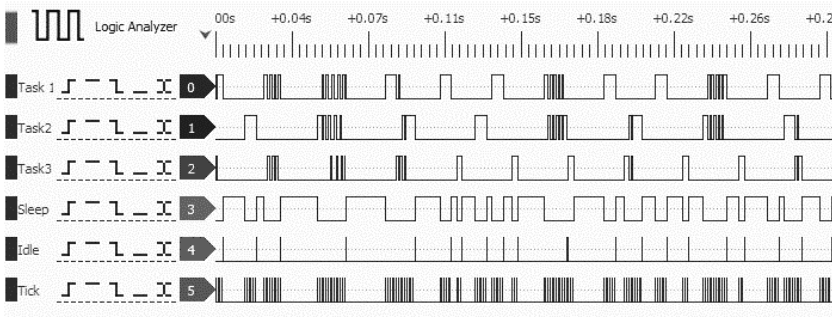


Рисунок 54. Коррекция задач для примера с переходом в режим tickles idle.

Теперь вы можете видеть, что периоды времени, в течении которых наш микроконтроллер реально имеет возможность перейти в режим сна, возникают весьма нерегулярно и имеют разную продолжительность. Мы уже говорили о том, что короткие промежутки сна не выгодны. Здесь и можно применить константу configEXPECTED_IDLE_TIME_BEFORE_SLEEP, описанную выше. Установим для нее значение 5.

```
#define configEXPECTED_IDLE_TIME_BEFORE_SLEEP 5
```

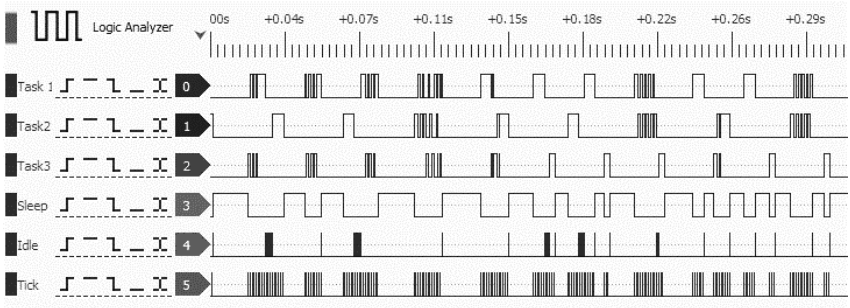


Рисунок 55. Корректный момент перехода в режим пониженного энергопотребления.

Теперь, как видите, короткие по времени промежутки сна исчезли и наш микроконтроллер переходит в режим пониженного энергопотребления только в те промежутки времени, когда это действительно целесообразно. Разумеется, в каждом конкретном проекте вам придется подбирать значение этой константы экспериментально.

А теперь давайте разберемся с тем, как за несколько строк кода получить такой замечательный результат. Как и отмечалось ранее нам необходимо только модифицировать конфигурационный файл FreeRTOSConfig.h добавив в него определение двух констант:

```
#define configUSE_TICKLESS_IDLE                 1
#define configEXPECTED_IDLE_TIME_BEFORE_SLEEP   5
```

Первая указывает на необходимость использовать реализацию tickles idle, предложенную портом FreeRTOS, вторая указывает на то, что не стоит переходить в состояние сна, если прогнозируемая продолжительность сна менее 5 (пяти) тиков.

Разумеется программисту интересно разобраться с макросом portSUPPRESS_TICKS_AND_SLEEP() немного подробнее. Посмотреть на то, как реализована предлагаемая портом процедура перехода в режим tickles idle. В качестве примера рассмотрим, как этот макрос определен для микроконтроллера STM32F40x. Файл portmacro.h содержит следующие строки:

```
/* Tickless idle/low power functionality. */
#ifndef portSUPPRESS_TICKS_AND_SLEEP
  extern void vPortSuppressTicksAndSleep(TickType_t xExpectedIdleTime);
  #define portSUPPRESS_TICKS_AND_SLEEP( xExpectedIdleTime )
```

```
          vPortSuppressTicksAndSleep( xExpectedIdleTime )
#endif
```

Здесь #define подменяет portSUPPRESS_TICKS_AND_SLEEP() на vPortSuppressTicksAndSleep(). Функция vPortSuppressTicksAndSleep() определяется здесь как внешняя (extern). Ввиду большого объёма кода не вижу смысла приводить ее на страницах книги. Читатель сможет найти код этой функции в файле port.c. Заметьте, что она обозначена как переопределяемая (__weak) и может быть заменена пользовательской реализацией без какой-либо модификации порта.

```
__weak void vPortSuppressTicksAndSleep( TickType_t xExpectedIdleTime )
```

Говоря о процедуре подготовки микроконтроллера к переходу в режим пониженного энергопотребления, необходимо понимать, что этот процесс занимает определенное время. Сначала планировщик определяет, что выполняются условия вызова portSUPPRESS_TICKS_AND_SLEEP(). Затем, уже в коде макроса происходит сохранение текущего времени, перенастройка источника прерываний и прочее. Существует шанс, что между моментом времени, когда было принято решение о переходе в режим сна и фактическим вызовом инструкции __wfi может произойти изменение состояния какой-либо задачи или возникнет аппаратное прерывание. Для того, чтобы своевременно отреагировать на подобные события существует функция API проверяющая статус системы непосредственно перед переходом в сон.

```
eSleepModeStatus eTaskConfirmSleepModeStatus( void );
```

Функция умеет возвращать одно из значений, перечисленных в файле task.h:

```
typedef enum
{
   eAbortSleep = 0,         /* Прервать вход в сон. */
   eStandardSleep,          /* Перейти в сон на указанное время. */
   eNoTasksWaitingTimeout   /* Перейти в сон до внешнего прерывания. */
} eSleepModeStatus;
```

Таким образом, перед тем, как внутри макроса portSUPPRESS_TICKS_AND_SLEEP() будет выполнен переход в состояние сна, мы должны вызвать eTaskConfirmSleepModeStatus() и получить обновленную информацию о статусе. Функция вернет значение eAbortSleep – если за время подготовки появилась какая-либо задача в состоянии готовности; eStandardSleep – если ничего не изменилось и микроконтроллер готов перейти в состояние сна на определенный промежуток времени; eNoTasksWaitingTimeout – если микроконтроллер можно перевести в состояние глубокого сна.

С отменой перехода в сон все понятно, это не более чем механизм для предотвращения перехода в сон, если за время подготовки к этому процессу произошло какое-либо прерывание и появилась какая-либо задача в состоянии «Готова». А вот возвращаемое значение eNoTasksWaitingTimeout открывает перед программистом достаточно любопытные возможности. Прежде всего, нужно знать, что функция eTaskConfirmSleepModeStatus() вернет значение eNoTasksWaitingTimeout только при выполнении следующих условий:

1. Программные таймеры не используются, планировщик не должен выполнять функцию обратного вызова таймера когда-либо в будущем.
2. Все задачи приложения находятся либо в приостановленном состоянии, либо в заблокированном состоянии с бесконечным таймаутом (значение таймаута portMAX_DELAY), в связи с этим планировщик не может определить время доступное для сна.

Также, код portSUPPRESS_TICKS_AND_SLEEP() содержит вызов двух макросов:

```
#define configPRE_SLEEP_PROCESSING      PreSleepProcessing
#define configPOST_SLEEP_PROCESSING     PostSleepProcessing
```

Один из них вызывается непосредственно перед, другой сразу после перехода в состояние с пониженным энергопотреблением. Взгляните на небольшой фрагмент кода portSUPPRESS_TICKS_AND_SLEEP():

```
configPRE_SLEEP_PROCESSING(xModifiableIdleTime);
if( xModifiableIdleTime > 0 )
{
  __dsb( portSY_FULL_READ_WRITE );
  __wfi();
```

```
  __isb( portSY_FULL_READ_WRITE );
}
configPOST_SLEEP_PROCESSING( xExpectedIdleTime );
```

Благодаря этим макросам вы можете выполнять собственные фрагменты кода и готовить периферийные блоки микроконтроллера к изменению режимов энергопотребления. При подготовке данной главы, описываемые макросы были использованы для визуализации процесса перехода в сон:

```
void PreSleepProcessing(uint32_t ulExpectedIdleTime)
{
   GPIOA->BSRR = GPIO_PIN_3;
}

void PostSleepProcessing(uint32_t ulExpectedIdleTime)
{
   GPIOA->BSRR = (uint32_t)GPIO_PIN_3 << 16U;
}
```

Корректировка времени

Как вы можете видеть, режим tickles idle может быть реализован и при необходимости модифицирован достаточно простыми средствами. Но у кажущейся простоты есть и скрытые проблемы. Проблема заключается в правильном учете времени. Нужно сказать, что мы уже не один раз обращались к этой проблеме. В режиме tickles idle проблема заключается в том, что пока микроконтроллер находится в режиме сна, все счетчики, включая счетчик тактов, остановлены. А значит, после возвращения планировщик ничего не будет знать о проведенном во сне времени, что приведет к нарушению работы таймаутов и программных таймеров, будут нарушены временные интервалы периодических задач.

Для решения этих проблем, частичного решения этих проблем и корректировки времени служит специальная функция API:

```
void vTaskStepTick( const TickType_t xTicksToJump );
```

Вызов функции vTaskStepTick() переводит счетчик тиков операционной системы на xTicksToJump тиков вперед. Грубо говоря, xTicksToJump это количество тактов RTOS, прошедших с момента

остановки прерывания. Для правильной работы параметр должен быть меньше или равен параметру переданному в макрос portSUPPRESS_TICKS_AND_SLEEP().

Ниже приведен краткий, упрощенный, шаблон позволяющий понять алгоритмы реализуемые внутри макроса portSUPPRESS_TICKS_AND_SLEEP(). Это может помочь как понять процессы происходящие в этом макросе, так и послужит шаблоном для разработки собственной реализации.

```
void vPortSuppressTicksAndSleep( TickType_t xExpectedIdleTime )
{
  unsigned long ulLowPowerTimeBeforeSleep,
                ulLowPowerTimeAfterSleep;
  eSleepModeStatus eSleepStatus;

  /* Сохраняем текущее время. */
  ulLowPowerTimeBeforeSleep = ulGetExternalTime();
  /* Останавливаем тиковые прерывания. */
  prvStopTickInterruptTimer();
  /* Запрещаем прерывания. */
  disable_interrupts();
  /* Проверяем можно ли безопасно переходить в режим сна. */
  eSleepStatus = eTaskConfirmSleepModeStatus();
  if( eSleepStatus == eAbortSleep )
  {
     /* Прерываем переход в сон. */
     prvStartTickInterruptTimer();
     enable_interrupts();
  }
  else
  {
    if( eSleepStatus == eNoTasksWaitingTimeout )
    {
       /* Переход в сон на неопределенно длительное время. */
       prvSleep();
    }
    else
    {
       /* Конфигурируем прерывание таймера черех xExpectedIdleTime. */
       vSetWakeTimeInterrupt( xExpectedIdleTime );
       /* Переход в режим сна. */
       prvSleep();
       /* Определяем время после пробуждения. */
       ulLowPowerTimeAfterSleep = ulGetExternalTime();
       /* Корректируем время ядра. */
       vTaskStepTick( ulLowPowerTimeAfterSleep –
                      ulLowPowerTimeBeforeSleep );
```

```
    }
    /* Возобновляем прерывания. */
    enable_interrupts();
    /* Активируем тиковый таймер. */
    prvStartTickInterruptTimer();
  }
}
```

Шаблон содержит отсылки к основной реализуемой логике, однако максимально абстрагирован от аппаратной реализации.

Глава 10. Мьютексы

В структуре этой книги мне очень хотелось, чтобы между темами использования семафоров и потокобезопасности прошло некоторое время. С одной стороны, в этих темах нет ничего сложного, с другой, для новичка это может представлять определенные сложности в понимании не самих механизмов, а в их необходимости.

По большей части говоря о мьютексах (англ. mutex) в операционной системе FreeRTOS мы говорим о еще одном, удобном и практичном механизме, который позволяет обеспечить взаимное исключение (англ. mutual exclusion) доступа к какому-либо ресурсу со стороны нескольких задач. Что позволяет предотвратить ситуации, когда две или более задач одновременно модифицируют один и тот же ресурс. Т.е. по сути мьютекс это модификация бинарного семафора наиболее пригодная для реализации потокобезопасности обьектов.

Преимущества мьютексов:

- Простота - легче использовать, чем семафоры.
- Предсказуемые результаты - исключает нежелательные ситуации гонки данных.
- Мьютексы занимают по 8 байт памяти при использовании простого (non-recursive) режима и 16 байт при использовании рекурсивного (recursive) режима.

Недостатки:

- Потенциальные перегрузки - если задача удерживает мьютекс слишком долго.
- Проблемы масштабируемости - возрастает число мьютексов с большим количеством задач.

Для того, чтобы функционал мьютексов стал доступен, необходимо в конфигурационном файле FreeRTOSConfig.h определить константу configUSE_MUTEXES и присвоить ей значение 1.

Рассмотрим процесс функционирования мьютексы более детально.

Мьютексы создаются функцией xSemaphoreCreateMutex(). Эта функция возвращает указатель на мьютекс, который в последствии передается в функции xSemaphoreTake() и xSemaphoreGive(). Мьютексы

могут быть рекурсивными (позволяют одной задаче получить доступ несколько раз) и нерекурсивными (доступ разрешен только одной задаче). Это определяется параметром uxRecursive и будет подробно рассмотрено позднее.

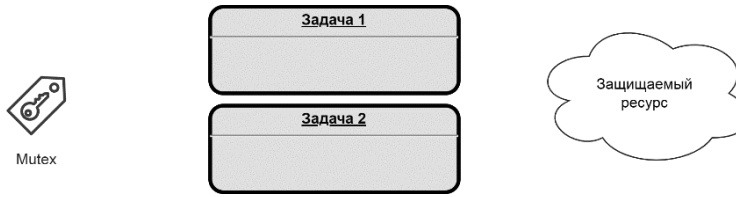


Рисунок 56. Конкурирующие задачи и защищаемый ресурс.

Существуют две задачи. Задача 1 и Задача 2, которые конкурируют за доступ к некоторому защищаемому мьютексом ресурсу.

Рисунок 57. Получение токена задачей 1.

Задача 1. Вызывает xSemaphoreTake() и пытается получить токен. Ей это удается и теперь она может получить доступ к защищаемому ресурсу.

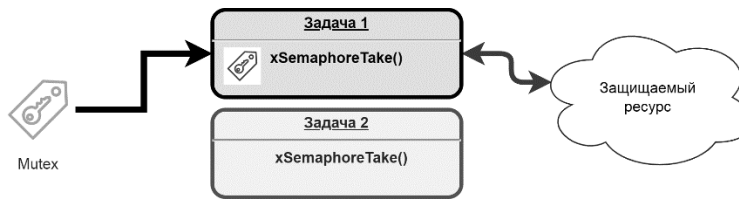


Рисунок 58. Попытка получения токена, который в данный момент занят.

При этом, если Задача 2, попытается получить токен, пока он занят и используется Задачей 1, она перейдет в состояние блокировки на время таймаута.

Рисунок 59. Освобождение токена.

Когда Задача 1 удовлетворит свои потребности по использованию защищаемого ресурса, она может вызвать функцию xSemaphoreGive() и вернуть токен.

Рисунок 60. Получение токена другой задачей.

В свою очередь, наличие токена приведет к тому, что Задача 2 выйдет из состояния блокировки и сможет получить токен для осуществления доступа к защищаемому ресурсу.

Рассмотрим использование мьютексов на простом примере. Две задачи будут конкурировать за доступ к глобальной переменной sharedResource. Одна из задач будет пытаться увеличить значение переменной, другая – уменьшить его.

```
#define MUTEX_HOLD_TIME pdMS_TO_TICKS(500)
SemaphoreHandle_t xSemaphore = NULL;
uint32_t sharedResource;
```

Определим величину таймаута, мьютекс и ресурс.

```
void vTask1(void pvParameters)
{
   if (xSemaphoreTake(xSemaphore, MUTEX_HOLD_TIME) == pdPASS)
   {
      sharedResource++;
      xSemaphoreGive(xSemaphore);
   }
```

```
}
```

Задача 1 будет пытаться увеличить значение переменной sharedResource.

```
void vTask2(void *pvParameters)
{
   if (xSemaphoreTake(xSemaphore, MUTEX_HOLD_TIME) == pdPASS)
   {
      sharedResource--;
      xSemaphoreGive(xSemaphore);
   }
}
```

Задача 2 будет стараться уменьшить значение этой переменной.

```
int main(void)
{
   xSemaphore = xSemaphoreCreateMutex();

   /* Создаем 2 задачи */
   xTaskCreate(vTask1, "Task 1", 128, NULL, tskIDLE_PRIORITY + 3, NULL);
   xTaskCreate(vTask2, "Task 2", 128, NULL, tskIDLE_PRIORITY + 3, NULL);

   /* Запуск FreeRTOS */
   vTaskStartScheduler();
}
```

Нам останется только запустить планировщик.

Создание мьютекса

Как следует из написанного выше, мьютекс это не более чем разновидность семафоров. Все семафоры в операционной системе FreeRTOS описываются единым типом дескриптора - SemaphoreHandle_t.

До того, как использование мьютекса станет возможным он должен быть создан функцией API xSemaphoreCreateMutex().

```
SemaphoreHandle_t xSemaphoreCreateMutex( void );
```

Проблемы использования мьютексов

Существует несколько потенциальных проблем, связанных с нарушением планируемого функционирования ОСРВ при использовании мьютексов. Эти проблемы достаточно сложно отлаживаемые.

Инверсия приоритета

Давайте детально рассмотрим одну из возможных проблем. В примере определены 2 задачи с разным приоритетом. Приоритет задачи 2 выше, чем задачи 1. Обе задачи используют мьютекс для защиты доступа к некоторому ресурсу.

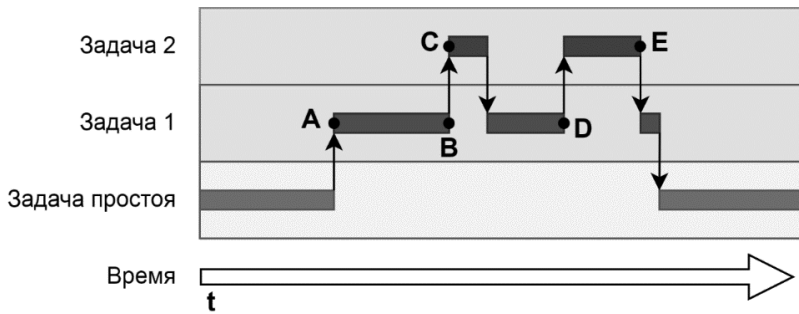


Рисунок 61. Инверсия приоритета.

(A) Задача 1 по какой-либо причине переходит из состояния блокировки в состояние выполняется. Тем самым происходит вытеснение задачи простоя. Задача 1 выполняется и получает токен, берет мьютекс для доступа к защищаемому ресурсу.

(B) из состояния блокировки выходит задача 2 имеющая приоритет выше приоритета задачи 1. Задача 2 вытесняет задачу 1.

(C) Задача 2 пытается взять токен, однако мьютекс по-прежнему удерживается задачей 1 пользующейся защищаемым ресурсом. В результате выполнения функции xSemaphoreTake, задача 2 переходит в состояние блокировки.

(D) Задача 1 заканчивает работу с защищаемым ресурсом и возвращает мьютекс. Автоматически высокоприоритетная задача 2 выходит из состояния блокировки и вытесняет задачу 1.

(E) Задача 2 завершает использование защищаемого ресурса, возвращает мьютекс и переходит в состояние блокировки, ожидая следующего времени выполнения. Это позволяет вернуться к выполнению Задачи 1, которая сразу после этого тоже переходит в состояние блокировки. В системе остается единственная задача, которая может выполняться – задача простоя.

Этот пример иллюстрирует очень интересную ситуацию, в которой Задача 2 с более высоким приоритетом должна ждать, пока Задача 1 с более низким приоритетом откажется от управления мьютексом. Таким образом, задача с более высоким приоритетом задерживается задачей с более низким приоритетом, что называется «инверсией приоритета».

И если вы думаете, что это самое плохое, что может случиться, то стоит взглянуть на следующий пример развития ситуации. В этом примере имеются три задачи. Задача с низким приоритетом, средним приоритетом и высоким приоритетом. Задача с низким приоритетом и задача с высоким приоритетом используют мьютекс для обеспечения безопасного доступа к разделяемому ресурсу.

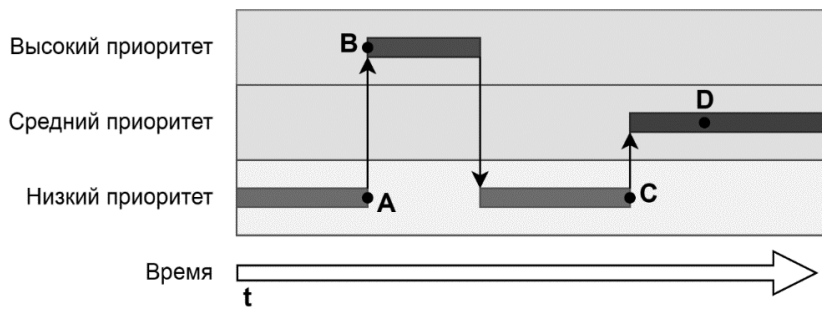


Рисунок 62. Проблемы с использованием мьютексов.

(A) Задача с низким приоритетом получила мьютекс и начала доступ к разделяемому ресурсу до того, как была вытеснена задачей с высоким приоритетом.

(B) Задача с высоким приоритетом попыталась получить мьютекс для доступа к разделяемому ресурсу. Эта попытка завершилась переходом в состояние блокировки т.к. мьютекс получен и используется задачей с самым низким приоритетом. Будет выполнен возврат к задаче с самым низким приоритетом.

(C) Задача с самым низким приоритетом была вытеснена задачей со средним приоритетом до того, как успела вернуть мьютекс.

(D) Задача со средним приоритетом выполняется, мьютекс не освобожден и это блокирует выполнение задачи с высоким приоритетом.

Проблема инверсии приоритета — это серьезная проблема. В небольших, встраиваемых системах эту проблему можно избежать на этапе проектирования системы анализируя то, как осуществляется доступ к разделяемым ресурсам.

Наследование приоритетов

Именно проблема инверсии приоритетов показывает основное отличие между мьютексами и двоичными семафорами. Мьютексы - это двоичные семафоры, которые включают механизм наследования приоритета. В то время как двоичные семафоры являются лучшим выбором для реализации синхронизации между задачами или между задачами и прерыванием, мьютексы являются лучшим выбором для реализации простого взаимного исключения доступа к какому-либо ресурсу.

Наследование приоритетов — это схема, уменьшающая негативные последствия инверсии приоритетов. Эта схема не «исправляет» инверсию приоритета, а просто уменьшает ее влияние, гарантируя, что инверсия всегда ограничена по времени. Однако наследование приоритетов усложняет анализ синхронизации системы, и полагаться на него для правильной работы системы не рекомендуется. Наследование приоритета работает путем временного повышения приоритета держателя мьютекса до приоритета задачи с наивысшим приоритетом, которая пытается получить тот же мьютекс. Задача с низким приоритетом, которая удерживает мьютекс, «наследует» приоритет задачи, ожидающей мьютекс.

Пожалуйста, учитывайте, что приложения, работающие в условиях жесткого реального времени должны быть спроектированы таким образом, чтобы инверсия приоритета не происходила.

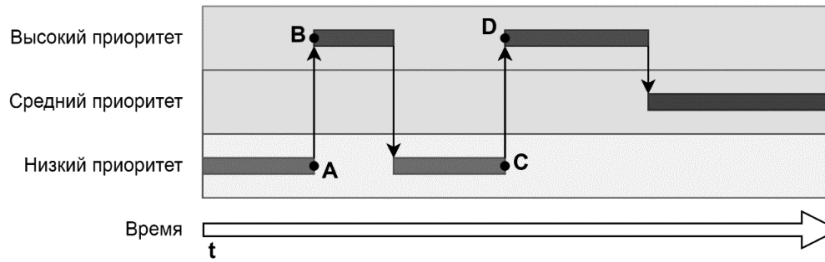


Рисунок 63. Наследование приоритетов.

(A) Задача с низким приоритетом получила мьютекс и начала доступ к разделяемому ресурсу до того, как была вытеснена задачей с высоким приоритетом.

(B) Задача с высоким приоритетом попыталась получить мьютекс для доступа к разделяемому ресурсу. Эта попытка завершилась переходом в состояние блокировки т.к. мьютекс получен и используется задачей с самым низким приоритетом. Будет выполнен возврат к задаче с самым низким приоритетом.

(C) Задача с низким приоритетом наследует максимальный приоритет ожидающей мьютекс задачи. Таким образом, задача не может быть вытеснена задачей со средним приоритетом и будет выполняться до тех пор, пока не освободит мьютекс и задача с высоким приоритетом не сможет выйти из состояния блокировки. После того, как задача вернет мьютекс она вернется к своему изначальному приоритету.

(D) Задача с высоким приоритетом получает мьютекс и доступ к разделяемому ресурсу.

На самом деле мы не просто так рассматриваем проблемы, возникающие с использованием такого механизма как мьютексы. Как только что было показано, функция наследования приоритетов влияет на приоритет задач, использующих мьютекс. И именно по этой причине мьютексы нельзя использовать в подпрограммах обслуживания прерываний.

Пат

«Пат» является еще одной потенциальной ловушкой, в которую можно попасть при использовании мьютексов для взаимного

исключения. В западной литературе авторы используют еще одно (как им кажется более литературное) название данной проблемы – «смертельные объятия».

Пат представляет собой проблему взаимной блокировки задач, которая возникает, когда две задачи не могут выполняться, потому что они обе ожидают ресурса, удерживаемого другой.

В качестве примера рассмотрим простой сценарий. Предположим, что существуют Задача 1 и Задача 2. Обе задачи используют мьютексы 1 и 2 для защиты используемых ресурсов.

1. В процессе выполнения Задача 1 получает Мьютекс 1.
2. Задача 1 вытесняется Задачей 2.
3. Задача 2 в ходе своего выполнения получает Мьютекс 2, пытается получить Мьютекс 1, но не может этого сделать т.к. Мьютекс 1 удерживается Задачей 1. Задача 2 вынуждено переходит в состояние блокировки.
4. Управление возвращается к задаче 1. Задача 1 пытается получить Мьютекс 2, но он уже получен Задачей 2. Задача 1 также переходит в состояние блокировки.

Как результат выполнения данного сценария мы оказываемся в ситуации, когда Задача 1 ожидает мьютекс удерживаемый Задачей 2, а Задача 2 ожидает мьютекс удерживаемый Задачей 1. Мы оказываемся в патовой ситуации и ни одна из задач не может продолжать свою работу.

Как и в случае с инверсией приоритета, лучший способ избежать взаимной блокировки — увидеть потенциальную возможность ее возникновения во время разработки. Спроектировать систему таким образом, чтобы исключить возникновение взаимных блокировок. В частности, нужно избегать ситуаций, когда какая-либо задача блокируется на неопределенно долгое время. Следует обязательно использовать таймаут. Используйте таймаут, немного превышающий ожидаемое максимальное время ожидания мьютекса. Тогда невозможность получить мьютекс в течение этого времени будет признаком ошибки проектирования, которая может привести к патовой ситуации.

Рекурсивные мьютексы

Пат, описанный в прошлом разделе. Когда вы читали его сценарий, не приходило ли вам в голову, что подобного исхода можно добиться

всего с одной задачей? Можете ли вы представить сценарий, при котором задача, использующая мьютекс, заблокирует сама себя? Давайте рассмотрим не совсем реалистичный, но возможный сценарий такой блокировки:

1. Задача А получает мьютекс.
2. В процессе работы Задача А вызывает функцию «В».
3. В коде функции «В» также используется использованный задачей мьютекс, что вызывает блокировку.

Как результат задача заблокирована в ожидании мьютекса, который сама же и удерживает.

Здесь мы можем наблюдать такое же лотание «дыр», как и в наследовании приоритетов. Разработчики предлагают бороться с данной схемой самоблокировки при помощи функционала рекурсивных мьютексов.

Одна и та же задача может «брать» рекурсивный мьютекс более одного раза. «Возврат» рекурсивного мьютекса должен быть выполнен для каждого получения мьютекса задачей. Напоминает модернизированные счетные семафоры.

Стандартные и рекурсивные мьютексы создаются и используются аналогичным образом:

- Стандартные мьютексы создаются с помощью xSemaphoreCreateMutex(). Рекурсивные мьютексы создаются с помощью xSemaphoreCreateRecursiveMutex(). Обе функции имеют схожие прототипы.
- Стандартный мьютекс можно «взяты» с помощью функции xSemaphoreTake(). Рекурсивные мьютексы с помощью xSemaphoreTakeRecursive().
- Стандартные мьютексы можно «отдать» с помощью xSemaphoreGive(). Рекурсивные мьютексы - xSemaphoreGiveRecursive().

Планирование задач

Давайте вернемся к вопросу политики планирования. Мы обсуждали ее в главе, посвященной вопросам функционирования задач. Если две задачи с разным приоритетом используют один и тот же

мьютекс, то политика планирования FreeRTOS четко определяет порядок выполнения задач. Задача с наивысшим приоритетом, которая может быть запущена, будет выбрана как задача, которая переходит в состояние «Выполняется». Например, если задача с высоким приоритетом находится в состоянии «Заблокировано» в ожидании мьютекса, удерживаемого задачей с низким приоритетом, то задача с высоким приоритетом вытеснит задачу с низким приоритетом, как только задача с низким приоритетом вернет мьютекс. В результате задача с высоким приоритетом станет держателем мьютекса. Именно этот пример мы детально рассматривали в разделе «Наследование приоритетов».

Однако достаточно часто делается неверное предположение относительно порядка выполнения задач, в том случае, когда задачи имеют одинаковый приоритет. Если Задача 1 и Задача 2 имеют одинаковый приоритет, Задача 1 находится в состоянии блокировки ожидания мьютекс, удерживаемый Задачей 2, то Задача 1 не будет вытеснять Задачу 2, когда Задача 2 «отдаст» мьютекс.

Вместо этого Задача 2 останется в состоянии «Выполняется», а Задача 1 просто перейдет из состояния «Заблокирована» в состояние «Готова». Рассмотрим этот сценарий подробнее.

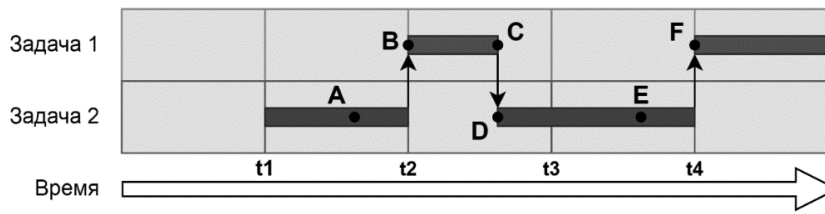


Рисунок 64. Нарушение справедливого распределения времени.

(A) Задача 2 выполняется, разделяя доступное время с Задачей 1 т.к. обе имеют одинаковый приоритет. Задача 2 «берет» мьютекс.

(B) Задача 1 переходит в состояние «Выполняется» получив свой квант доступного времени.

(C) Задача 1 пытается получить мьютекс и переходит в состояние блокировки ожидая освобождения мьютекса Задачей 2.

(D) Задача 2 получает управление не потому, что пришло ее время, а потому, что Задача 1 перешла в состояние блокировки.

(E) Задача 2 возвращает мьютекс и тем самым создает условие для разблокировки Задачи 1. Однако Задача 1 и Задача 2 имеют одинаковый приоритет. В связи с этим фактическое переключение контекста произойдет только после истечения выделенного задаче кванта времени.

(F) Задача 1 получает управление в начале следующего тикового периода.

Важным для понимания является именно то, что в приведенном сценарии будет нарушено привычное справедливое распределение доступного процессорного времени между задачами с одинаковым приоритетом. Почему это происходит? Есть две основные причины:

1. Если принимать во внимание, что используется режим разделения времени (константа конфигурации configUSE_TIME_SLICING имеет значение 1), а Задачи 1 и Задача 2 имеют одинаковый приоритет, переключение между задачами происходит только в момент тикового прерывания.
2. Если же задача использует мьютекс в «тесном» цикле, и переключение контекста происходит каждый раз, когда задача «отдает» мьютекс, то задача будет оставаться в состоянии выполнения только в течение короткого времени. Если две или более задач используют один и тот же мьютекс в тесном цикле, то время обработки будет потрачено впустую из-за быстрого переключения между задачами.

Из этого следует, что, если мьютекс используется в тесном цикле более чем одной задачей, и задачи, использующие мьютекс, имеют одинаковый приоритет, необходимо позаботиться о том, чтобы задачи получали примерно равное количество времени обработки.

Тесный цикл (англ. "tight loop") - это программный цикл, который выполняется многократно в течение короткого промежутка времени, обычно без задержек и с минимальной нагрузкой на процессор. Тесные циклы часто используются в программировании для обработки ввода/вывода или для выполнения вычислительных задач, которые требуют повторения одной и той же операции с высокой частотой. Примером тесного цикла может быть следующий фрагмент кода на C:

```
while (1) {
   // выполнение каких-то операций
}
```

Задача привратник

Задача привратника (англ. "Gatekeeper task") - это задача в многозадачной операционной системе, которая контролирует доступ других задач к различным ресурсам системы, таким как процессорное время, память, дисковое пространство и т.д.

Задача привратника является основным компонентом механизма защиты операционной системы от ошибок и злоумышленных действий других задач. Она принимает решения о том, какие задачи могут получить доступ к ресурсам системы и на каком уровне приоритета, а какие задачи должны быть ограничены или отклонены.

Задача привратника обычно работает вместе с механизмом управления доступом, который определяет права доступа для каждой задачи в системе. Вместе они обеспечивают безопасность и защиту операционной системы от ошибок и злоумышленных действий. Наиболее ярким примером задачи привратника может служить механизм управления доступом в ОС Linux, который контролирует доступ к файловой системе, сетевым ресурсам, устройствам ввода-вывода и другим ресурсам системы. Задача привратник обеспечивает чистый метод реализации взаимного исключения без риска инверсии приоритета или взаимоблокировки.

На самом деле это очень интересная концепция. Рассматривая вопросы потокобезопасности мы уже познакомились с примером сложностей, которые возникают во время того, как несколько различных задач в силу той или иной необходимости пытаются использовать один и тот же ресурс. Для того, чтобы избавиться от сложностей, эта глава предлагает мьютексы, привратник – как альтернатива, хорошая альтернатива мьютексам. Рассмотрим на простейшем примере:

```
static void prvStdioGatekeeperTask( void *pvParameters )
{
   char *pcMessageToPrint;
   /* Это единственная задача, которой разрешено взаимодействовать
      со стандартным выводом. */
   for( ;; )
   {
      /* Ожидаем сообщений из очереди. */
      xQueueReceive( xPrintQueue, &pcMessageToPrint, portMAX_DELAY );
      printf( "%s", pcMessageToPrint );
```

```
        fflush( stdout );
    }
}
```

Придерживаясь данной концепции, единственный способ что-либо вывести в стандартный вывод - это отправить сообщение в очередь задачи привратника. Да, это не всегда удобный и не всегда оправданный вариант решения проблем с потокобезопасностью, но мы должны иметь его в арсенале наших средств.

Глава 11. Группы событий

Говоря о таком механизме как группы событий (англ. Event Groups) мы возвращаемся к вопросам обеспечения механизма синхронизации между задачами. По сути большинство из того, чем мы занимаемся на протяжении последних глав этой книги - это вопросы синхронизации между задачами. И не так важно, говорим мы о синхронизации между обработчиками прерываний и задачами, на которые отложена обработка, или о синхронизации задач для обеспечения требуемой последовательности выполнения алгоритмов обработки информации, мы занимаемся синхронизацией.

В отличие от ранее описанных средств группы событий позволяют реализовывать гораздо более сложные схемы. Более того, это единственный механизм, который приближает нас к реализации многих, специфичных для логических автоматов и программируемых логических контроллеров функций без дополнительного кода.

Чем же группы событий отличаются от ранее описанных семафоров и очередей? Существуют два важных отличия:

1. Группы событий позволяют задаче пребывать в состоянии блокировки в ожидании пока не произойдет комбинация одного или нескольких событий.
2. Группа событий разблокирует все задачи, которые ожидали одного и того же события или комбинации событий.

Т.е. можно говорить о «широковещательной» рассылке событий более чем одной задаче. Механизм позволяет задаче ожидать в заблокированном состоянии любого из набора событий. И если ваш проект содержал большое количество семафоров, то группы событий дают возможность уменьшить объем оперативной памяти, используемой приложением, поскольку группа может заменить множество двоичных семафоров.

Функционал событий не является стандартным компонентом. Для использования событий вам необходимо включить исходный файл FreeRTOS event_groups.c в состав вашего проекта.

Группы, флаги, биты

В понимании темы группы событий нет ничего сложного. Основная единица - «Флаг» события - логическое значение (0 или 1), используемое для указания того, произошло событие или нет. Отсюда, группа событий — это набор флагов событий.

Флаг может принимать только одно из значений, 0 или 1, что позволяет хранить состояние флага события в одном бите, а состояние всех флагов группы в одной переменной. Состояние каждого флага события в группе событий представлено одним битом в переменной типа EventBits_t.

Если флаг события в группе событий установлен в значение логической 1, то мы можем сказать, что событие произошло. И наоборот, если флаг события имеет значение логического 0, мы можем утверждать, что это событие не произошло.

Группа событий, как и следует из написанного выше - это не более чем переменная типа EventBits_t.

Рисунок 65. Биты в группе событий.

Таким образом, бит 0 это флаг 0, а бит 23, это флаг 23.

Давайте в качестве примера рассмотрим группу событий, у которой установлены флаги 1, 3, 8, 9, 14. В этом случае значение переменной группы событий будет равно 0x430A (b00000000 01000011 00001010).

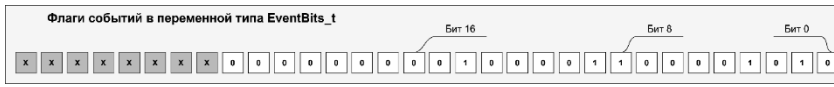


Рисунок 66. Значение переменной соответствующей группе событий.

В виду того, что FreeRTOS портирована на большинство архитектур, количество битов событий в группе событий может изменяться и зависит от константы конфигурации configUSE_16_BIT_TICKS, определяемой в конфигурационном файле FreeRTOSConfig.h.

- Если configUSE_16_BIT_TICKS равно 1, то каждая группа событий содержит 8 используемых битов событий.
- Если configUSE_16_BIT_TICKS равно 0, то каждая группа событий содержит 24 используемых бита событий.

Группа событий — это самостоятельный объект, доступ к которому может получить любая задача или обработчик прерывания. Любое количество задач может устанавливать биты в одной и той же группе событий, и любое количество задач может считывать биты из одной и той же группы событий.

Создание группы событий

Как и все элементы ядра операционной системы группа событий должна быть явным образом создана до того, как будет использована.

Для того, чтобы ссылаться на группу событий используют переменную типа EventGroupHandle_t. API-функция xEventGroupCreate() используется для создания группы событий и в качестве результата своей работы возвращает дескриптор EventGroupHandle_t соответствующий создаваемой группе событий.

```
EventGroupHandle_t xEventGroupCreate( void );
```

Функция возвращает дескриптор группы событий. Значение дескриптора отличное от нуля означает что группа успешно создана и может быть использована. Если возвращается NULL, группа событий не может быть создана, так как для нее недостаточно памяти в куче.

Установка событий

В предыдущих главах мы уже рассматривали вопрос о том, что объекты использующиеся для синхронизации выполнения задач, как правило, имеют два варианта функции API. Функция, предназначенная для работы в контексте операционной системы и функция, предназначенная для выполнения в коде обработчика прерываний. Для работы с флагами событий также предусмотрены два функции. По

традиции начнем с варианта, который следует использовать в контексте операционной системы.

```
EventBits_t xEventGroupSetBits( EventGroupHandle_t xEventGroup,
                                const EventBits_t uxBitsToSet );
```

Для функции определены следующие параметры:

xEventGroup - дескриптор группы событий, для которой устанавливаются биты. Дескриптор группы событий возвращается функцией xEventGroupCreate(), используемой для создания группы событий.

uxBitsToSet - битовая маска, указывающая бит (биты) события, которые нужно установить в группе событий. При установке битов применяется операция побитового ИЛИ аргументами которого выступают существующее значение группы событий и биты переданные в uxBitsToSet.

Возвращаемое значение - текущее состояние группы событий на момент возврата из функции xEventGroupSetBits().

Говоря о функционале объектов, использующихся для синхронизации задач, мы, например, можем уверенно сказать, что предоставление семафора является детерминированной операцией, поскольку заранее известно, что предоставление семафора может привести к переходу не более чем одной задачи из состояния «Заблокировано» в состояние «Готова». В отношении группы событий мы не можем определить заранее сколько конкретно задач выйдет из состояния блокировки, по этой причине установка бит в группе событий не является детерминированной операцией.

Подходы, примененные при реализации дизайна операционной системы FreeRTOS, не позволяют осуществлять не детерминированные операции внутри кода обработчика прерываний. По этой причине мы можем утверждать, что для реализации функционала xEventGroupSetBitsFromISR() - функции устанавливающей биты в группе событий разработчики «лотали дыры» в собственной концепции. Дело в

том, что функция xEventGroupSetBitsFromISR() не устанавливает биты события непосредственно внутри обработчика прерывания, вместо этого она откладывает выполнение действия на задачу демона RTOS аналогично тому, как это делают функции API программных таймеров.

Функция, используемая для выполнения в теле обработчика прерывания, имеет следующий прототип:

```
BaseType_t xEventGroupSetBitsFromISR( EventGroupHandle_t xEventGroup,
                                      const EventBits_t uxBitsToSet,
                                      BaseType_t *pxHigherPriorityTaskWoken );
```

Для функций определены следующие параметры:

xEventGroup - дескриптор группы событий, для которой устанавливаются биты. Дескриптор группы событий возвращается функцией xEventGroupCreate(), используемой для создания группы событий.

uxBitsToSet - битовая маска, указывающая бит (биты) события, которые нужно установить в группе событий. При установке битов применяется операция побитового ИЛИ аргументами которого выступают существующее значение группы событий и биты переданные в uxBitsToSet.

pxHigherPriorityTaskWoken - Если задача демона находилась в состоянии блокировки и ожидала, пока в очереди команд таймера появятся новые данные, выполнение xEventGroupSetBitsFromISR() приведет к разблокировке демона. Если приоритет задачи демона выше приоритета выполняемой в данный момент задачи (задачи, которая была прервана обработчиком прерывания), то *pxHigherPriorityTaskWoken будет установлен в значение pdTRUE.

	Если xEventGroupSetBitsFromISR() устанавливает это значение в pdTRUE, то перед выходом из обработчика прерывания необходимо вызвать переключение контекста.
Возвращаемое значение	- pdPASS - команда «выполнить функцию» была успешно передана в очередь команд таймера. pdFAIL - команда «выполнить функцию» не может быть помещена в очередь команд таймера, потому что очередь команд таймера уже заполнена.

Пожалуйста, обратите внимание на несколько моментов связанных с функциями установки битов группы событий:

- Возвращаемое функцией установки бит группы событий xEventGroupSetBits() значение не всегда будет равно устанавливаемому значению в связи с тем, что на момент возврата из функции группа событий уже может оказаться модифицированной какой-либо разблокированной задачей.
- Функции со схожим функционалом xEventGroupSetBits() и xEventGroupSetBitsFromISR() имеют различные возвращаемые значения.

Ожидание событий

Как было анонсировано, есть функция, которая позволяет задаче ожидать наступления любого события в группе. Функция xEventGroupWaitBits() позволяет задаче считывать значение группы событий и при необходимости ожидать в состоянии блокировки, пока один или несколько битов события в группе событий не будут установлены.

```
EventBits_t xEventGroupWaitBits( const EventGroupHandle_t xEventGroup,
                                 const EventBits_t uxBitsToWaitFor,
                                 const BaseType_t xClearOnExit,
                                 const BaseType_t xWaitForAllBits,
                                 TickType_t xTicksToWait );
```

Параметры данной функции несколько сложнее, чем то, что мы использовали ранее:

xEventGroup - дескриптор группы событий, содержащей считываемые биты события.

uxBitsToWaitFor - битовая маска, указывающая бит или биты события для проверки в группе событий. Например, если задаче нужно протестировать установку битов 0 и 2 в группе событий, для uxBitsToWaitFor следует установить значение 0x05 (b0101).

xClearOnExit - если xClearOnExit = pdTRUE и условие заданное параметром uxBitsToWaitFor (маска) выполнено, то заданные маской биты событий будут сброшены в 0 до того, как задача выйдет из функции xEventGroupWaitBits(). Если xClearOnExit = pdFALSE, то состояние битов событий в группе событий функцией xEventGroupWaitBits() не изменяется.

xWaitForAllBits - если xWaitForAllBits = pdFALSE, то задача, перешедщая в состояние блокировки (для ожидания выполнения условия) выйдет из блокировки, когда любой бит, указанный в uxBitsToWaitFor, станет установленным или истечет время таймаута, установленное xTicksToWait.
Если xWaitForAllBits = pdTRUE, то задача, перешедшая в состояние блокировки, выйдет из него только после того, как будут установлены все биты, указанные в uxBitsToWaitFor, или, истечет время таймаута.

xTicksToWait - время таймаута в течение которого задача может находиться в состоянии блокировки ожидая выполнения условия разблокировки. Время таймаута указано в тиках. Используйте макрос pdMS_TO_TICKS() для преобразования

	времени. Установка xTicksToWait = portMAX_DELAY приведет к тому, что задача будет ожидать максимально возможное в данной архитектуре время.
Возвращаемое значение	- функция xEventGroupWaitBits() возвращает значение группы событий, само значение зависит от того, было выполнено условие разблокировки или произошел таймаут. В случае, когда было выполнено условие разблокировки, возвращается значение группы событий на момент выполнения условия разблокировки вызывающей задачи (до того, как какие-либо биты были автоматически очищены, если xClearOnExit было pdTRUE). Т.е. возвращаемое значение = условие разблокировки. Если же истекло время таймаута, то возвращаемое значение является значением группы событий на момент истечения времени блокировки. В этом случае возвращаемое значение не будет соответствовать условию разблокировки.

В виду очевидной важности функции xEventGroupWaitBits() для организации логических алгоритмов средствами FreeRTOS, разберемся с логикой ее параметров несколько подробнее, чем это описано выше.

Некоторая сложность определения того, войдет ли задача в состояние «Заблокировано» и того, когда она покинет это состояние вызвана тем фактом, что «условие разблокировки» в данной функции задается комбинацией значений параметров uxBitsToWaitFor и xWaitForAllBits из них:

- uxBitsToWaitFor указывает, какие биты событий в группе событий нужно тестировать.
- xWaitForAllBits указывает, следует ли использовать побитовое «ИЛИ» или побитовое «И». Задача не перейдет в состояние блокировки, если ее условие разблокировки выполнено во время вызова xEventGroupWaitBits().

Несмотря на то, что эти два параметра достаточно логичны, практическое использование порой вызывает некоторые сложности. Рассмотрим логику работы на простых примерах:

Значение группы	uxBitsToWaitFor	xWaitForAllBits	Результат
0000	0101	pdFALSE	Задача будет заблокирована. В группе событий нет ни одного бита из ожидаемых.
0000	0101	pdTRUE	Задача будет заблокирована. В группе событий нет ни одного бита из ожидаемых.
0100	0110	pdFALSE	Задача не будет заблокирована, xWaitForAllBits = pdFALSE, и один из двух битов в uxBitsToWaitFor, уже установлен в группе событий.
0100	0110	pdTRUE	Задача будет заблокирована, т.к. xWaitForAllBits имеет значение pdTRUE в то время, как только один из двух битов, указанных в uxBitsToWaitFor установлен в группе событий.

Также хотелось бы обратить ваше внимание на общую логику использования группы событий. Как уже отмечалось, существует большая разница между семафорами, очередями и группой событий. Прежде всего она проявляется в том, что флаги группы событий могут быть доступны и использованы для синхронизации неопределенного количества задач. Иными словами, в состоянии блокировки могут находиться одновременно несколько задач в ожидании одного и того же бита в группе событий. С появлением этого бита неопределенное

количество задач перейдет из состояния блокировки в состояние готовности. Это является достаточно интересным инструментом, о котором можно задуматься в поисках средств групповой синхронизации. Но как происходит процесс очистки (сброса) бита события?

Одним из очевидных способов очистки бита событий, который уже выполнил свою функцию в оповещении задачи о произошедшем событии является параметр xClearOnExit функции xEventGroupWaitBits(). Если xClearOnExit = pdTRUE, то биты событий будут очищены в ходе выполнения функции. Из-за того, что проверка и очистка битов проводится и осуществляется внутри единой функции для вызывающей задачи эта операция является атомарной операцией и не приведет к возникновению потенциальных сложностей.

Между тем, существует и возможность очистки событий посредством вызова функции xEventGroupClearBits().

Практика

Этот раздел не просто так назван кратким словом «Практика». Описанного ранее функционала, в том числе и функционала задействованного непосредственно в синхронизации задач, более чем достаточно для построения широкого спектра прикладного программного обеспечения. Но группы событий очень сильно отличаются от ранее описанных объектов. Они на самом деле уникальны и открывают перед нами возможности с минимальным количеством кода решать достаточно сложные логические задачи.

При разработке логических контроллеров или систем управления нам не всегда требуется решать сложные математические задачи. Иногда необходимо обеспечить выполнение задач простой комбинационной логики. Например, представим себе контроллер разрабатываемый для нужд ЖКХ. Контроллер имеет несколько входов для подключения внешних датчиков: датчик открытия двери, датчик сухого хода насоса, датчик снижения давления в магистрали, датчик наличия 3 фазного напряжения питания и т.п. и несколько выходов формирующих управляющие воздействия. Предположим, что нам требуется включать сигнальную лампу каждый раз, когда происходит любое событие – изменение состояние входа. Включать насос при падении давления,

отключать его при поступлении сигнала сухого хода и так далее. В подобных применениях логика может меняться. И именно инструмент группы событий в операционной системе FreeRTOS позволяет нам реализовать гибкий механизм изменения логики без дополнительного кода.

Рассмотрим это на примере. Создадим задачу 2, которая будет находиться в состоянии блокировки ожидая события связанного с 1-м битом в группе. Основную часть времени эта задача будет проводить в состоянии блокировки, выходя их него только при наступлении обозначенного события. Как видите из переданных параметров задача сбрасывает выставленный бит события.

```
void StartTask02(void const * argument)
{
  for(;;)
  {
     xEventGroupWaitBits(myGroupHandle, 0x01, 0x01,
                         pdTRUE, portMAX_DELAY);
     __NOP();
  }
}
```

Задача 3 по аналогии с задачей 2 тоже будет находиться в заблокированном состоянии, но она будет ожидать наступления события, ассоциированного с битом 2.

```
void StartTask03(void const * argument)
{
  for(;;)
  {
     xEventGroupWaitBits(myGroupHandle, 0x02, 0x02,
                         pdTRUE, portMAX_DELAY);
     __NOP();
  }
}
```

И, наконец, задача 4 должна будет реагировать на комбинацию битов 1 и 2 в группе событий. Причем для перехода задачи в состояние готовности потребуется присутствие обоих битов в группе. Тем самым мы можем проверить основной функционал группы событий и отследить работу задач уже привычным нам методом.

```
void StartTask04(void const * argument)
{
  for(;;)
  {
     xEventGroupWaitBits(myGroupHandle, 0x03, 0x03,
                         pdTRUE, portMAX_DELAY);
     __NOP();
  }
}
```

Для формирования событий создадим периодическую задачу 1. Каждые 2 миллисекунды она будет формировать события в группе событий последовательно помещая туда коды 0x01, 0x02, 0x03, что соответствует 1-му, 2-му и комбинации 1+2 битов событий в группе.

```
void StartTask01(void const * argument)
{
  for(;;)
  {
     xEventGroupSetBits(myGroupHandle, 0x01);
     osDelay(2);
     xEventGroupSetBits(myGroupHandle, 0x02);
     osDelay(2);
     xEventGroupSetBits(myGroupHandle, 0x03);
     osDelay(2);
  }
}
```

Не сложно догадаться, что результатом выполнения описанного кода будет следующий график на логическом анализаторе.

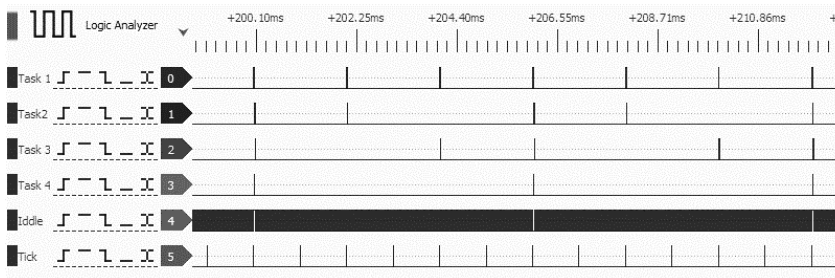


Рисунок 67. Группа событий. Практический пример.

Как видно из графика Задачи 2, 3 и 4 работают как ожидалось. Они выходят из состояния блокировки и выполняются при появлении

соответствующей комбинации событий. Таким образом мы можем реализовать реакцию подмножества задач в ответ на гибкое изменение событий в группе. Что позволит нам реализовывать достаточно сложные логические схемы и алгоритмы.

Проблемы точки синхронизации

При изучении семафоров буквально в двух словах затрагивался вопрос взаимной синхронизации подмножества задач. Для реализации этого с использованием семафоров придется создавать достаточно большое количество семафоров, а главное – жестко продумывать их логику. Т.е. мы должны будем разработать некоторый план взаимной синхронизации задач и реализовать его в виде семафоров, обеспечивающих выполнение задач в нужной последовательности.

Группу событий также можно использовать для создания точки синхронизации. Это реализуется тремя практическими шагами:

- Каждой задаче, которая должна участвовать в синхронизации, назначается уникальный бит события в группе событий.
- Каждая задача устанавливает свой собственный бит события, когда достигает точки синхронизации.
- Установив свой собственный бит события, задача блокируется в группе событий, ожидая установки битов события, которые представляют все другие синхронизирующие задачи.

Это выглядит весьма перспективным, но функции xEventGroupSetBits() и xEventGroupWaitBits() нельзя использовать в этом сценарии. Если бы они использовались, то установка бита (подтверждение того, что задача достигла своей точки синхронизации) и проверка битов (определение, достижение другими задачами точки синхронизации) выполнялись бы как две отдельные операции.

Для того, чтобы понять проблему использования функций xEventGroupSetBits() и xEventGroupWaitBits() при создании точки синхронизации для трех задач рассмотрим простой сценарий в котором три задачи пытаются синхронизироваться используя группу событий:

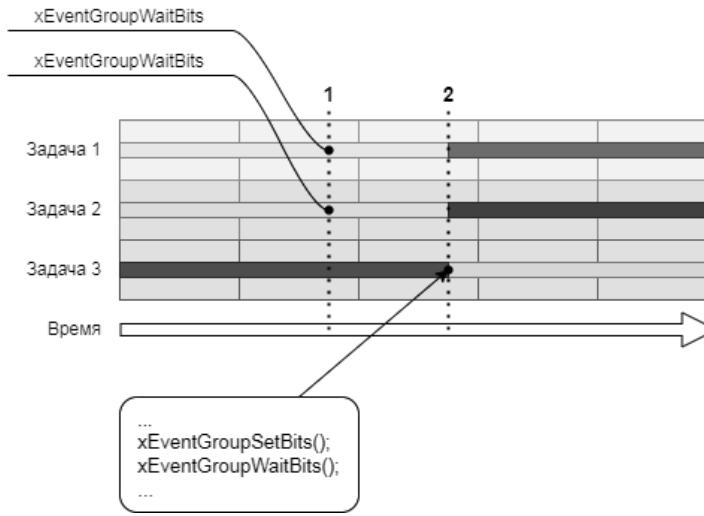


Рисунок 68. Проблемы точки синхронизации.

1. Задачи 1 и 2 достигли точки синхронизации. Их биты событий установлены в группе событий. Задачи находятся в состоянии блокировки, ожидая пока Задача 3 также достигнет точки синхронизации и установит свой бит события.

2. Задача 3 достигает точки синхронизации и использует функцию xEventGroupSetBits() для установки своего бита в группе событий. Как только бит задачи 3 установлен, задачи 1 и 2 выходят из состояния блокировки, считая, что синхронизация достигнута. Сбрасывают все три бита событий. И только после этого Задача 3 вызывает xEventGroupWaitBits(), чтобы дождаться установки всех трех битов события, но к тому времени все три бита события уже очищены, задачи 1 и 2 оставили свои соответствующие точки синхронизации.

Показанный сценарий наглядно иллюстрирует, какие проблемы могут возникнуть при использовании группы событий как точки синхронизации задач. Очевидно, что причина этих проблем кроется только в том, что мы вынуждены использовать связку из двух функций выполняемых последовательно. Это xEventGroupSetBits() и xEventGroupWaitBits(). Для устранения проблем при синхронизации разработчики FreeRTOS создали специальную функцию API - xEventGroupSync().

Создание точки синхронизации

Функция xEventGroupSync() позволяет двум и более задачам использовать группу событий как инструмент синхронизации друг с другом. Задача может установить неопределенное количество бит событий в группе, а затем, находясь в состоянии блокировки, ожидать комбинации битов события. Эти действия выполняются в виде одной непрерывной операции.

```
EventBits_t xEventGroupSync(EventGroupHandle_t xEventGroup,
                            const EventBits_t uxBitsToSet,
                            const EventBits_t uxBitsToWaitFor,
                            TickType_t xTicksToWait );
```

xEventGroup - дескриптор группы событий, содержащей считываемые биты события.

uxBitsToSet - битовая маска, указывающая бит или биты события, которые нужно установить в группе событий. Значение группы событий обновляется путем побитового «ИЛИ» существующего значения группы событий со значением, переданным в uxBitsToSet. Например, установка для uxBitsToSet значения 0x04 (двоичный код 0100) приведет к установке бита события 3 (если он еще не был установлен), в то время как все остальные биты событий в группе событий останутся без изменений.

uxBitsToWaitFor - битовая маска, указывающая бит или биты события для проверки в группе событий. Например, если вызывающая задача хочет дождаться установки битов события 0, 1 и 2 в группе событий, установите для uxBitsToWaitFor значение 0x07 (двоичный 111).

xTicksToWait - максимальное время таймаута, в течение которого задача должна оставаться в заблокированном состоянии, чтобы дождаться

выполнения условия разблокировки. Время таймаута указывается в тиках. Макрос pdMS_TO_TICKS() можно использовать для преобразования времени, указанного в миллисекундах, во время, указанное в тиках.

Возвращаемое значение

- если функция xEventGroupSync() возвращает результат из-за того, что было выполнено условие разблокировки, то возвращаемое значение является значением группы событий на момент выполнения условия разблокировки вызывающей задачи (до того, как какие-либо биты были автоматически очищены до нуля). В этом случае возвращаемое значение также будет соответствовать условию разблокировки вызывающей задачи. Если функция xEventGroupSync() возвращает значение из-за того, что время блокировки, указанное параметром xTicksToWait, истекло, то возвращаемое значение является значением группы событий на момент истечения времени блокировки. В этом случае возвращаемое значение не будет соответствовать условию разблокировки вызывающей задачи.

Глава 12. Уведомления

Как и любой удачный проект, FreeRTOS развивался и претерпевал определенные изменения. В версии V1.2.0, которая увидела свет в 2002 году, была представлена функциональность семафоров. Тогда API семафоров базировалось на целом наборе макросов, в свою очередь вызывавших функции работы с очередями. С одной стороны, такой необычный вариант дизайна имел определенное преимущество, ведь он добавлял функциональность без увеличения объёма кода. Для 2002 года это было достаточно важно, ведь микроконтроллеры не могли похвастаться столь большими размерами flash и оперативной памяти. С другой стороны, недостатков у такого дизайна тоже хватало. Основываясь на очередях, семафоры были нетипично «тяжелыми» объектами за счет наследования лишних особенностей очередей. Но главное, они работали медленно.

С тех пор, коллектив, работавший над FreeRTOS, искал все более оптимальные варианты развития операционной системы. Так постепенно сформировался функционал, особенности и набор API всех основных элементов ядра операционной системы.

Раздумывая над дизайном прикладного программного обеспечения, использующего FreeRTOS, вы наверняка задумывались над тем, что каждый элемент ядра - очереди, семафоры, мьютексы, группы событий, таймеры, - все они используют оперативную память для размещения служебной информации. Они все обходятся разработчику в определенное количество байт оперативной памяти. И все они по своей сути являются коммуникативными объектами, берущими на себя функции по передаче информации между задачами.

Единственным способом прямого общения между задачами был и остается – Уведомление (англ. Notifications). В ранних версиях FreeRTOS у задач было только одно уведомление. А начиная с FreeRTOS V.10.4.0, каждая задача имеет массив уведомлений. Каждое уведомление содержит 32-разрядное значение и логическое состояние, которые вместе занимают всего 5 байт оперативной памяти.

В сентябре 2020 года появилась статья, написанная Ричардом Барри «Уменьшение объема оперативной памяти и ускорение выполнения с помощью уведомлений FreeRTOS». Эта статья предваряет собой

изменение курса всей команды разработчиков относительно использования уведомлений в дальнейшем развитии FreeRTOS.

Основное преимущество уведомлений заключается в том, что для их функционирования не используются коммуникационные объекты. Уведомление отправляется от задачи к задаче напрямую. А, следовательно, и работает быстрее.

По умолчанию функционал уведомлений отключен. Функциональность уведомлений о задачах не является обязательной. Для активации необходимо определить configUSE_TASK_NOTIFICATIONS присвоив константе значение 1. Если функционал уведомлений активирован, каждая задача получает «Состояние уведомления», которое может иметь два состояния «Ожидает» или «Не Ожидает», и «Значение уведомления», которое представляет собой 32-разрядное целое беззнаковое число. Когда задача получает уведомление, ее состояние уведомления меняется на «Ожидает». Когда задача считывает уведомления, ее состояние уведомления меняется на «Не Ожидает». Задача может быть заблокирована с необязательным таймаутом в ожидании уведомления.

Информация об уведомлениях хранится в TCB (Task Control Block) каждой созданной в FreeRTOS задачи. Давайте посмотрим на то, какая вообще информация хранится в TCB.

```
typedef struct tskTaskControlBlock
{
        volatile StackType_t     *pxTopOfStack;
        ListItem_t               xStateListItem;
        ListItem_t               xEventListItem;
        UBaseType_t              uxPriority;
        StackType_t              *pxStack;
        char                     pcTaskName[ configMAX_TASK_NAME_LEN ];
        UBaseType_t              uxBasePriority;
        UBaseType_t              uxMutexesHeld;
        TaskHookFunction_t       pxTaskTag;

        #if( configUSE_TASK_NOTIFICATIONS == 1 )
                volatile uint32_t ulNotifiedValue;
                volatile uint8_t ucNotifyState;
        #endif

} tskTCB;
```

Как видите, директива условий компиляции #if модифицирует состав TCB включая в его состав информацию о уведомлениях.

Нет смысла расписывать все преимущества использования уведомлений. В конечном счете все сводится к двум очень важным вещам – увеличение скорости и экономия ресурсов. Что действительно необходимо внимательно рассмотреть – это ограничения, с которыми придется столкнуться при использовании уведомлений в реальных проектах. Рассмотрим случаи, в которых использование функционала уведомлений невозможно:

- Отправка информации в обработчик прерывания. Коммуникационные объекты, созданные как элемент ядра операционной системы можно использовать для обмена событиями и данными между задачами и обработчиками прерываний. Уведомления можно использовать только для отправки информации из обработчика прерывания в задачу, но не наоборот.
- Уведомлять более одной задачи одновременно. Доступ к коммуникационному объекту может получить любая задача или обработчик прерывания, которым известен дескриптор коммуникационного объекта. Любое количество задач и обработчиков прерывания может обрабатывать события или данные коммуникационного объекта. Уведомления отправляются непосредственно задаче-получателю, поэтому могут быть обработаны только задачей, которой отправлено уведомление.
- Уведомления не позволяют буферизировать данные. Например, очередь — это коммуникационный объект, который может содержать более одного элемента данных одновременно. Данные, отправленные в очередь, но еще не полученные из очереди, буферизуются внутри объекта очереди. Уведомления отправляют данные задаче перезаписывая ранее отправленное уведомление если оно еще не было получено задачей.
- Уведомления не позволяют осуществлять групповую синхронизацию. Например, группа событий — это коммуникационный объект, который можно использовать для одновременной отправки события более чем одной задаче. Уведомления о задачах отправляются непосредственно

принимающей задаче, поэтому могут быть обработаны только принимающей задачей.

- Отсутствует таймаут отправки информации. Если коммуникационный объект временно находится в состоянии, которое означает, что в него нельзя больше записывать данные или события (например, когда очередь заполнена, в очередь нельзя больше отправлять данные), тогда задачи, пытающиеся записать в объект, могут войти в состояние «Заблокировано», чтобы дождаться завершения операции записи. Если задача пытается отправить уведомление задаче, для которой уже есть уведомление, то задача-отправитель не может ждать в состоянии «Заблокировано», пока задача-получатель сбросит свое состояние уведомления.

Использование уведомлений

Уведомления — очень мощная функция, которую часто можно использовать вместо двоичного или счетного семафоров, группы событий и иногда даже очереди. Широкий спектр сценариев использования может быть достигнут с помощью функции xTaskNotify() для отправки уведомления задаче и функции xTaskNotifyWait() для получения уведомления.

Первой хорошей новостью является тот факт, что уведомление – не коммуникационный объект и нам ничего не нужно создавать или каким-либо образом инициализировать до того, как начать использовать.

Более того, в большинстве случаев полная гибкость, обеспечиваемая функциями xTaskNotify() и xTaskNotifyWait(), не требуется, и достаточно более простых функций. Например, функция xTaskNotifyGive() предоставляет собой более простую, но менее гибкую альтернативу xTaskNotify(), а ulTaskNotifyTake(), менее гибкая альтернатива xTaskNotifyWait().

Отправка уведомлений

Функция xTaskNotifyGive() отправляет уведомление задаче и увеличивает значение уведомления принимающей задачи. Как следует из

описанного функционала данная функция является прекрасной альтернативой бинарному семафору.

```
BaseType_t xTaskNotifyGive( TaskHandle_t xTaskToNotify );
```

xTaskToNotify - дескриптор задачи, которой отправляется уведомление.

Возвращаемое значение - единственное возможное возвращаемое значение – pdPASS.

Логично предположить, что у данной функции есть и специальная версия для вызова из обработчика прерываний:

```
void vTaskNotifyGiveFromISR( TaskHandle_t xTaskToNotify,
                             BaseType_t *pxHigherPriorityTaskWoken );
```

xTaskToNotify - дескриптор задачи, которой отправляется уведомление.

pxHigherPriorityTaskWoken - если задача, которой отправляется уведомление, ожидает получения уведомления в состоянии блокировки, то отправка уведомления приведет к тому, что задача выйдет из состояния блокировки. Если вызов vTaskNotifyGiveFromISR() приводит к тому, что задача выходит из состояния блокировки и ее приоритет выше, чем приоритет выполняемой в данный момент задачи (задачи, которая была прервана), то xHigherPriorityTaskWoken будет присвоено значение pdTRUE. Соответственно перед выходом из обработчика прерывания мы должны запросить переключение контекста.

Получение уведомлений

Функция ulTaskNotifyTake() позволяет задаче ожидать (в состоянии блокировки) пока ее значение уведомления не станет больше нуля. Функция уменьшает на единицу значение уведомления задачи. Задача перейдет в состояние блокировки, если счетчик уведомлений станет равным нулю.

В настоящее время команда разработчиков операционной системы FreeRTOS рекомендует использование этой функции как более простую и быструю замену бинарным и счетным семафорам.

```
uint32_t ulTaskNotifyTake( BaseType_t xClearCountOnExit,
                           TickType_t xTicksToWait );
```

xClearCountOnExit - если xClearCountOnExit = pdTRUE, то значение уведомления вызывающей задачи будет очищено до нуля перед возвратом вызова ulTaskNotifyTake(). Если xClearCountOnExit = pdFALSE и значение уведомления вызывающей задачи больше нуля, то значение уведомления будет уменьшено.

xTicksToWait - максимальное значение таймаута в течении которого задача будет находиться в заблокированном состоянии ожидая уведомления.

Возвращаемое значение - значение уведомления вызывающей задачи до того, как оно было очищено до нуля или уменьшено.

Уведомления как семафоры

Попробуем предложенный функционал уведомлений в качестве семафоров. Для этого создадим 4 задачи. Задача 1 будем иметь приоритет выше нормального и будет находиться в ожидании уведомления от обработчика прерывания.

```
void StartTask01(void *argument)
```

```
{
  for(;;)
  {
     ulTaskNotifyTake(0, portMAX_DELAY);
     __NOP();
  }
}
```

А задачи 2, 3 и 4 будут иметь нормальный приоритет и будут поровну делить все доступное процессорное время. Их код ничем не отличается от большинства примеров.

```
void StartTask02(void *argument)
{
  for(;;)
  {
     __NOP();
  }
}

void StartTask03(void *argument)
{
  for(;;)
  {
     __NOP();
  }
}

void StartTask04(void *argument)
{
  for(;;)
  {
     __NOP();
  }
}
```

Источником прерываний в нашем примере будет служить прерывание от таймера.

```
void TIM3_IRQHandler(void)
{
  BaseType_t xHigherPriorityTaskWoken;
  xHigherPriorityTaskWoken = pdFALSE;

  HAL_TIM_IRQHandler(&htim3);

  vTaskNotifyGiveFromISR( myTask01Handle, &xHigherPriorityTaskWoken );
  portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
```

```
}
```

В результате выполнения данного кода на логическом анализаторе мы получаем диаграмму, показанную на рисунке 69.

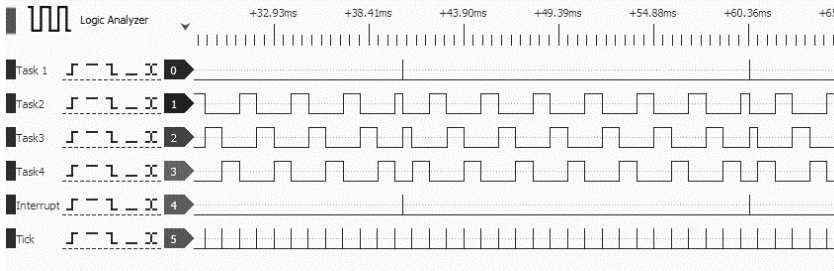


Рисунок 69. Пример использования уведомлений в качестве семафоров.

Здесь видны все основные элементы. Задачи 2, 3 и 4 разделяющие доступное время, обработчик прерывания и деблокируемая им задача 1. Остается только выяснить, как много времени проходит с момента возникновения прерывания и до того момента, как планировщик передаст управление задаче 1.

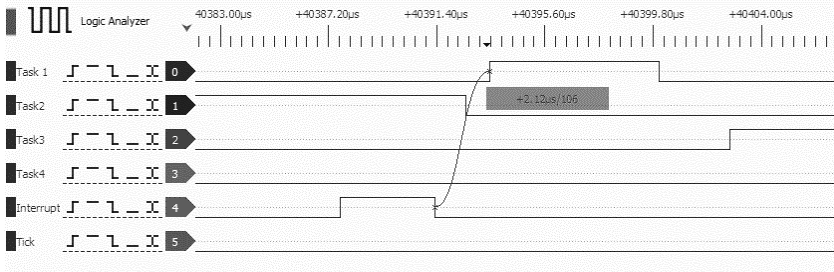


Рисунок 70. Скорость работы уведомлений.

От момента завершения работы обработчика прерывания и до передачи управления задаче 1 прошло 2,12 микросекунды. При этом, напомню, микроконтроллер работает на частоте 72 мГц.

Также весьма интересным является опробовать работу уведомлений в качестве замены счетных семафоров. Немного модифицируем наш пример. Теперь в обработчике прерывания от таймера мы будем посылать задаче 1 не одно, а три уведомления.

```
void TIM3_IRQHandler(void)
{
  BaseType_t xHigherPriorityTaskWoken;
  xHigherPriorityTaskWoken = pdFALSE;

  HAL_TIM_IRQHandler(&htim3);

  vTaskNotifyGiveFromISR( myTask01Handle, &xHigherPriorityTaskWoken );
  vTaskNotifyGiveFromISR( myTask01Handle, &xHigherPriorityTaskWoken );
  vTaskNotifyGiveFromISR( myTask01Handle, &xHigherPriorityTaskWoken );

  portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

После завершения работы такого обработчика прерывания счетчик уведомлений задачи 1 должен увеличиться на три уведомления. Остальной код примера оставим без изменений и посмотрим на то, как изменилась диаграмма, отображаемая логическим анализатором.

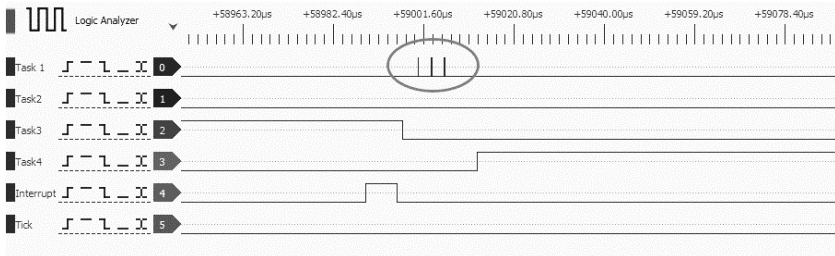


Рисунок 71. Счетчик уведомлений.

Как вы можете видеть, после завершения работы обработчика прерывания, задача 1 выполняет не 1, а 3 цикла считывая уведомления до тех пор, пока счетчик уведомлений не станет нулевым. После этого задача переходит в заблокированное состояние в ожидании нового уведомления.

Уведомления

Функция xTaskNotify() — это более мощная версия ранее рассмотренной xTaskNotifyGive(). Функция с расширенным функционалом позволяет решать несколько более широкий круг задач и более активно взаимодействовать с уведомляемыми задачами:

- Функции xTaskNotify() и xTaskNotifyGive() идентично увеличивают счетчик уведомлений, но xTaskNotify() позволяет взаимодействовать с отдельными битами уведомлений что позволяет использовать уведомления в качестве более легкой и быстрой альтернативы группам событий.
- Позволяет записывать произвольное число в значение уведомлений принимающей задачи, но только в том случае, если принимающая задача прочитала свое значение уведомления с момента последнего обновления. Это позволяет значению уведомления задачи обеспечивать функциональность, подобную той, что обеспечивается очередью, длина которой равна единице.
- Позволяет записывать произвольное число в значение уведомлений принимающей задачи, даже если принимающая задача не прочитала свое значение уведомления с момента последнего обновления. Это позволяет значению уведомления задачи обеспечивать функциональность, подобную той, что предоставляется API-функцией xQueueOverwrite(). Результирующее поведение иногда называют «почтовым ящиком».

```
BaseType_t xTaskNotify( TaskHandle_t xTaskToNotify,
                        uint32_t ulValue,
                        eNotifyAction eAction );
```

xTaskToNotify - дескриптор задачи, которой отправляется уведомление.

ulValue - см. описание ниже.

eAction - см. описание ниже.

Возвращаемое значение - возвращаемое значение pdPASS (за исключением действия eSetValueWithoutOverwrite)

Функция xTaskNotifyFromISR() — это версия xTaskNotify(), которую нужно использовать в коде обработчиков прерывания, и поэтому она имеет дополнительный параметр pxHigherPriorityTaskWoken.

```
BaseType_t xTaskNotifyFromISR( TaskHandle_t xTaskToNotify,
                               uint32_t ulValue,
                               eNotifyAction eAction,
                               BaseType_t *pxHigherPriorityTaskWoken );
```

Параметры функции xTaskNotify() и их влияние на результат работы функции:

eNoAction - Состояние уведомления для принимающей задачи установлено как ожидающее без обновления значения уведомления. Параметр xTaskNotify() ulValue не используется. Действие eNoAction позволяет использовать уведомление о задаче как более быструю и легкую альтернативу двоичному семафору.

eSetBits - Значение уведомления принимающей задачи объединяется операцией побитового «ИЛИ» со значением, переданным в параметре xTaskNotify() ulValue. Например, если для ulValue задано значение 0x06 (двоичный код 0110), то в значении уведомления будут установлены биты 1 и 2. Действие eSetBits позволяет использовать уведомление о задаче как более быструю и легкую альтернативу группе событий.

eIncrement - Значение уведомления принимающей задачи увеличивается. Параметр xTaskNotify() ulValue не используется. Действие eIncrement позволяет использовать уведомление о задаче как более быструю и легкую альтернативу двоичному или счетному семафору и эквивалентно более простой API-функции xTaskNotifyGive().

eSetValueWithoutOverwrite - Если перед вызовом xTaskNotify() у принимающей задачи было ожидающее уведомление, то никаких действий не предпринимается, и xTaskNotify() возвращает pdFAIL. Если у принимающей задачи не было ожидающих уведомления до вызова xTaskNotify(), то значение уведомления принимающей задачи устанавливается равным значению, переданному в параметре xTaskNotify() ulValue.

eSetValueWithOverwrite - Значение уведомления принимающей задачи устанавливается равным значению, переданному в параметре xTaskNotify() ulValue, независимо от того, было ли у принимающей задачи ожидающее уведомление до вызова xTaskNotify() или нет.

Ожидание уведомлений

Фнукция xTaskNotifyWait() — более мощная версия ulTaskNotifyTake(). Эта функция позволяет задаче находиться в состоянии блокировки с необязательным тайм-аутом. xTaskNotifyWait() предоставляет опции для очистки битов в значении уведомления вызывающей задачи как при входе в функцию, так и при выходе из функции.

```
BaseType_t xTaskNotifyWait( uint32_t ulBitsToClearOnEntry,
                            uint32_t ulBitsToClearOnExit,
                            uint32_t *pulNotificationValue,
                            TickType_t xTicksToWait );
```

ulBitsToClearOnEntry - Если у вызывающей задачи не было ожидающего уведомления до того, как она вызвала xTaskNotifyWait(), то любые биты, установленные в ulBitsToClearOnEntry, будут очищены в значении уведомления задачи при входе в функцию. Например, если ulBitsToClearOnEntry имеет значение 0x01, бит 0 значения уведомления задачи будет очищен. В качестве другого примера, установка для ulBitsToClearOnEntry значения 0xffffffff приведет к очистке всех битов в значении уведомления задачи, эффективно очистив их значение до 0.

ulBitsToClearOnExit - Если вызывающая задача выходит из функции xTaskNotifyWait() из-за того, что она получила уведомление, или из-за того, что уведомление уже было ожидающим при вызове xTaskNotifyWait(), то все биты, установленные в ulBitsToClearOnExit, будут очищены в значении уведомления задачи до того, как задача выйдет из функции xTaskNotifyWait( ). Установка для ulBitsToClearOnExit значения 0xffffffff очистит все биты в значении уведомления задачи.

***pulNotificationValue** - Используется для передачи значения уведомления задаче. Значение, скопированное

	в *pulNotificationValue, является значением уведомления задачи, каким оно было до того, как какие-либо биты были очищены параметром ulBitsToClearOnExit. pulNotificationValue является необязательным параметром и может быть установлен в NULL, если он не используется.
xTicksToWait	- таймаут, в течение которого вызывающая задача должна оставаться в состоянии «Заблокировано», чтобы дождаться перехода уведомления в состояние ожидания.
Возвращаемое значение	- Возможны два возвращаемых значения: 1. pdTRUE - указывает на то, что функция xTaskNotifyWait() возвращена из-за того, что было получено уведомление, или потому что вызывающая задача уже имела ожидающее уведомление при вызове xTaskNotifyWait(). 2. pdFALSE - указывает на то, что xTaskNotifyWait() возвратил вызов без получения уведомления о задаче. Если xTicksToWait не равен нулю, то вызывающая задача будет удерживаться в состоянии Blocked, ожидая, пока ее состояние уведомления станет ожидающим, но указанное время блокировки истекло до того, как это произошло.

Еще один пример

К этому моменту опыт читателя уже позволяет оценить все достоинства системы уведомлений. Очевидно, что она должна работать гораздо быстрее, чем привычные нам коммуникационные объекты. Это легко иллюстрирует простой пример. Предположим, что в создаваемом нами приложении существует обработчик прерывания от аналого-цифрового преобразователя. Вызов обработчика происходит каждый раз, когда ADC заканчивает преобразование. Обработчик должен считать полученное значение и отправить данные выбранной задаче на

дальнейшую обработку. По сути это очень похоже на сценарий отложенной обработки прерывания, когда мы часть работы переносим на задачу, выполняющуюся в контексте операционной системы.

В общем виде обработчик будет иметь следующий вид:

```
void ADC_ConversionEndISR( xADC *pxADCInstance )
{
   uint32_t ulConversionResult;
   BaseType_t xHigherPriorityTaskWoken = pdFALSE, xResult;

   /* Читаем преобразованное ADC значение. */
   ulConversionResult = ADC_low_level_read( pxADCInstance );
   /* Отправляем уведомление содержащее значение задаче vADCTask(). */
   xResult = xTaskNotifyFromISR( xADCTaskToNotify, ulConversionResult,
                                 eSetValueWithoutOverwrite,
                                 &xHigherPriorityTaskWoken );

   configASSERT( xResult == pdPASS );
   portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

И, для полноты картины, рассмотрим задачу, на которую отложена обработка полученной информации:

```
void vADCTask( void *pvParameters )
{
   uint32_t ulADCValue;
   BaseType_t xResult;

   for( ;; )
   {
     /* Ожидаем уведомления */
     xResult = xTaskNotifyWait( 0, 0, &ulADCValue,
                                xADCConversionFrequency * 2 );
     if( xResult == pdPASS )
     {
        ProcessADCResult( ulADCValue );
     }
     else
     {
        /* Мы не получили уведомление с данными за отведенное время */
     }
   }
}
```

Функции API предлагаемые для функционала уведомлений перекрывают весь диапазон сценариев. Они интуитивно понятны и могут быть использованы как замена классическим объектам обмена данных и синхронизации с учетом определенных, накладываемых уведомлениями ограничений.

Глава 13. Отладка и трассировка

С первых минут, освоения языка программирования каждый из нас начинает приобретать опыт по отладке и трассировке создаваемого кода. Если рассматривать этот процесс упрощенно, то все сводится к тому, что, наблюдая за ходом выполнения написанного нами кода, мы пытаемся выявить ситуации, когда он работает не так, как это ожидалось при его написании. Чаще всего, мы используем точки останова, для того, чтобы убедиться в том, что программа дошла до того или иного фрагмента кода и правильно выполнила логические операторы. Либо мы считываем значения переменных и наблюдаем за тем, правильно ли происходят вычисления в отлаживаемом коде.

Эти приемы отладки прекрасно работают в классической схеме организации кода. В действительности вы можете наблюдать и срабатывание таймера, и функционирование системы обработки прерываний и переход к тем или иным функциям. Но привычный опыт отладки часто дает сбой при попытке работы с операционной системой.

Сложность отладки приложений, использующих ОС, чаще всего связывают с невозможностью контроля за тем, когда переключается контекст и управление переходит к другой задаче. Грубо говоря, ход выполнения любой задачи может быть прерван в любой момент и управление будет передано другому фрагменту кода. Напрямую управлять этим процессом без вмешательства в код приложения мы не можем, а вот почва для возникновения глупых логических ошибок тут весьма благотворная. Разумеется, спрос рождает предложения, а инструментария для отладки за время существования FreeRTOS создано не мало. Глобально его можно разделить на две большие группы:

1. Инструментарий встроенный в IDE и сторонние утилиты.
2. Встроенные средства отладки операционной системы.

Стороннее Программное обеспечение

Многие годы популярности операционной системы на рынке принесли свои плоды. Различными компаниями создано большое количество разнообразного программного обеспечения по конфигурированию и отладке приложений, использующих операционную систему. Это все общие фразы, а в реальности мы нуждаемся всего в нескольких инструментах:

1. Генераторы кода, облегчающие интеграцию FreeRTOS в создаваемый проект и берущие на себя все вопросы, связанные с генерацией минимально необходимого шаблона кода.
2. Средства мониторинга. Программное обеспечение и средства отладки, позволяющие наблюдать за процессами переключения задач, функционированием объектов ядра, процессами синхронизации. Проще говоря, способ проверить, соответствует ли реальный процесс ожидаемому.
3. Средства оценки производительности. Некий механизм, позволяющий нам оценить реальную производительность системы и принять решение о необходимости или достаточности оптимизации.

Как видите, наши потребности в инструментарии простираются от планирования до отладки. В этом разделе мы кратко познакомимся с существующими решениями. Кратко потому, что описание каждого из упоминаемых продуктов достойно отдельной, объёмной, статьи.

Генератор кода

Как следует из названия, генератор кода - это инструмент, который позволяет автоматически создавать код. Он призван значительно ускорить процесс разработки, сократить количество ошибок и повысить качество создаваемого кода.

Как и любой другой инструмент, генератор кода не может решить все проблемы, стоящие перед разработчиком, но может существенно повысить скорость его труда за счет оптимизации и стандартизации некоторых рутинных операций:

- Настройка периферийных устройств: инициализация периферийных блоков микроконтроллера, таких как таймеры,

прерывания, АЦП, ЦАП и т.д. Помимо очевидной экономии времени мы освобождаемся от досадных ошибок, связанных с инициализацией периферийных блоков;
- Управление конфигурацией: большинство генераторов кода могут управлять конфигурацией подключаемых библиотек;
- Создание шаблонов: некоторые генераторы кода могут формировать шаблоны кода стандартных операций. Это могут быть как функции PID регулирования, так и фильтрация, взаимодействие с периферией.

Если говорить об использовании генераторов кода для построения шаблона приложения, использующего операционную систему FreeRTOS, то прежде всего ожидается, что генератор добавит необходимые файлы исходного кода операционной системы к проекту, использует правильные файлы порта, в той или иной мере сгенерирует файл конфигурации и, возможно, создаст шаблоны для задач.

Если разработчиков спросить о том, какие генераторы кода он знает для архитектуры ARM, то большинство назовет программу STM32CubeMX, но это далеко не единственный продукт такого рода:

1. STM32CubeMX - графический инструмент, который позволяет легко настроить периферийные устройства микроконтроллера STM32 и сгенерировать код для различных сред разработки.
2. Atmel START - онлайн-инструмент для настройки микроконтроллеров Atmel/Microchip и генерации кода для различных сред разработки.
3. NXP MCUXpresso Config Tools - графический инструмент, который позволяет настроить периферийные устройства микроконтроллера NXP и сгенерировать код для различных сред разработки.
4. Keil MDK - интегрированная среда разработки, которая включает в себя генератор кода для микроконтроллеров на ядре ARM.
5. IAR Embedded Workbench - интегрированная среда разработки, которая также включает в себя генератор кода для микроконтроллеров на ядре ARM.

Впечатляющий список. Наверное, не имеет смысла спорить с тем, что STM32CubeMX не просто так занимает первую строчку в этом списке. Это специализированный и глубоко проработанный продукт, не просто

настраивающий периферийные блоки, но и способный интегрировать в проект десятки различных сторонних библиотек. В том числе и FreeRTOS.

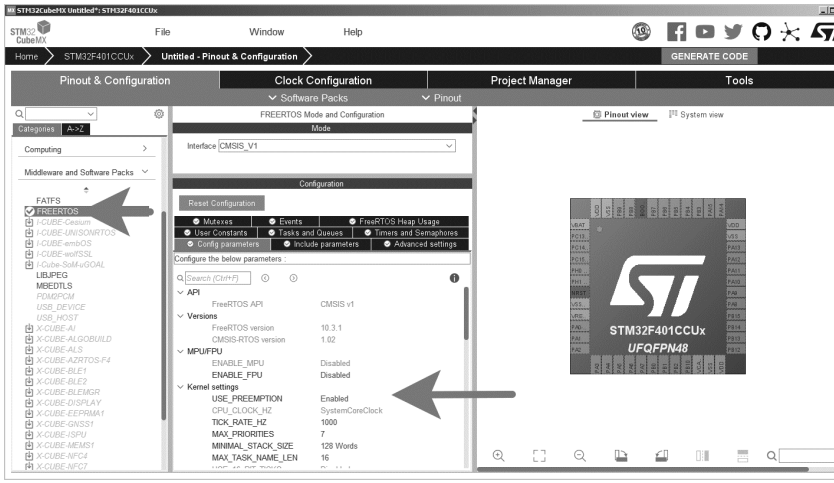


Рисунок 72. Экран программы STM32CubeMX

STM32CubeMX позволяет разработчику создать конфигурационный файл FreeRTOSConfig.h, настроить схему использования кучи, создать шаблоны задач и коммуникационных элементов ядра. Грубо говоря, единственное, что необходимо сделать разработчику, это написать код задач.

```
/* Private variables -----------------------------------------------*/
osThreadId myTask02Handle;
```

Дескриптор задачи

```
/* Private function prototypes -----------------------------------------*/
void StartTask02(void const * argument);
```

Прототип задачи

```
  /* definition and creation of myTask02 */
  osThreadDef(myTask02, StartTask02, osPriorityNormal, 0, 128);
  myTask02Handle = osThreadCreate(osThread(myTask02), NULL);
```

Код создания задачи

```
/* USER CODE END Header_StartTask02 */
void StartTask02(void const * argument)
{
  /* USER CODE BEGIN StartTask02 */
  /* Infinite loop */
  for(;;)
  {
    osDelay(1);
  }
  /* USER CODE END StartTask02 */
}
```

И, наконец, функция, воплощающая в себе задачу.

Весь этот объём кода был создан STM32CubeMX автоматически по заложенным в него шаблонам, очевидно, это экономит не мало времени. С определенными изменениями аналогично работают и другие генераторы. Но автоматически создаваемый код — это только самое начало процесса разработки. В дальнейшем потребности разработчиков меняются.

Мониторинг и отладка

Ожидаемо, что с развитием проекта возникает вполне обоснованная потребность в анализе хода выполнения задач, анализе алгоритма переключения контекста, синхронизации. Требуется некоторый механизм, отслеживающий логику работы операционной системы. Для этих целей в код FreeRTOS включены механизмы сбора статистики и отладочной информации. Мы подробнейшим образом рассмотрим их функционал несколько дальше в этой главе. Но, механизмы сбора статистики не имеют возможностей к визуализации. И в этом вопросе доступный инструментарий для мониторинга тоже радует.

Прежде всего, минимальный набор инструментария для отладки включен в состав практически всех современных IDE. Например, STM32CubeIDE имеет инструменты для анализа задач, очередей, семафоров и программных таймеров:

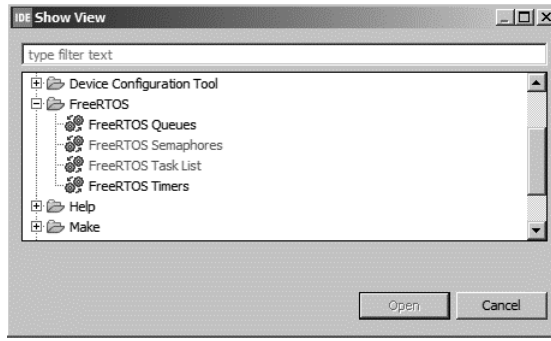
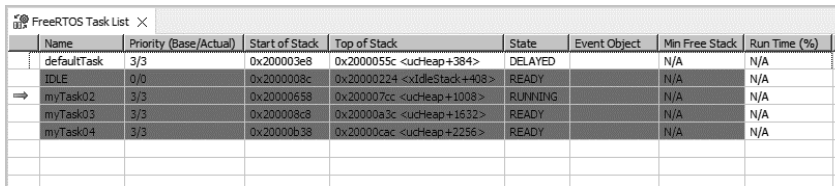


Рисунок 73. Доступные инструменты отладки.

Инструментарий по своему функционалу стоило бы отнести к базовому, в нем отсутствует возможность обновления данных в режиме реального времени, но определенный интерес как средство отладки этот инструмент представляет.



Name	Priority (Base/Actual)	Start of Stack	Top of Stack	State	Event Object	Min Free Stack	Run Time (%)
defaultTask	3/3	0x200003e8	0x2000055c <ucHeap+384>	DELAYED		N/A	N/A
IDLE	0/0	0x2000008c	0x20000224 <xIdleStack+408>	READY		N/A	N/A
⇒ myTask02	3/3	0x20000658	0x200007cc <ucHeap+1008>	RUNNING		N/A	N/A
myTask03	3/3	0x200008c8	0x20000a3c <ucHeap+1632>	READY		N/A	N/A
myTask04	3/3	0x20000b38	0x20000cac <ucHeap+2256>	READY		N/A	N/A

Рисунок 74. STM32CubeIDE просмотр состояния задач.

Несколько больший функционал по мониторингу представляется приложением STM32CubeMonitor. Этот пакет базируется на известном проекте Node-RED и позволяет в режиме реального времени, используя ARM-DAP, получать отладочную информацию.

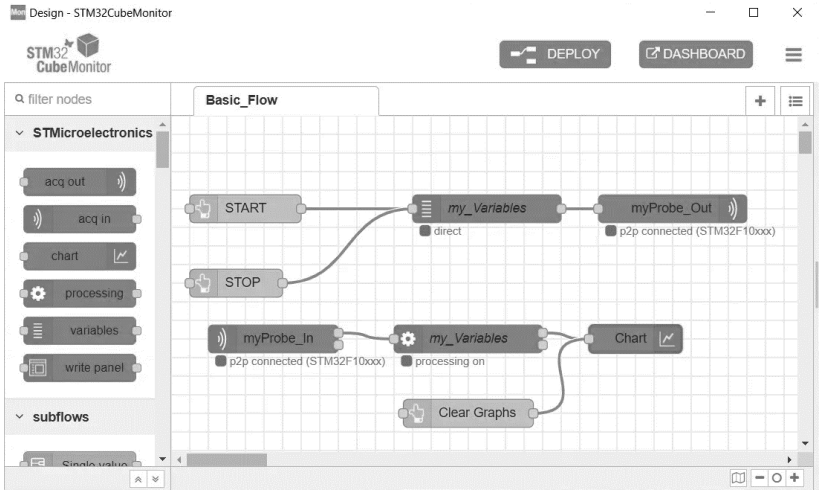


Рисунок 75. Окно программы STM32CubeMonitor.

Одним из существенных недостатков STM32CubeMonitor можно считать то, что он использует базовый функционал программатора-отладчика ST-Link со всеми присущими последнему недостатками и ограничениями. Это делает работу программы достаточно медлительной. С другой стороны, этого вполне достаточно для отображения некоторой долговременной статистики, накопительных результатов. Например, информации о наличии свободной памяти, нагрузке на ядро и т.п.

Но как поступить, если сложность реализуемого проекта требует использования более быстрых и информативных средств? В этом случае идеальным является использование специальной версии сборки FreeRTOS, именуемой - FreeRTOS+Trace. Это инструмент, который позволяет анализировать данные и визуализировать их в виде графиков и диаграмм. Он может помочь выявить узкие места в системе и оптимизировать ее работу. Происходит это за счет использования Tracealyzer, программного пакета, анализирующего работу FreeRTOS.

Основой этого проекта является коммерческий инструмент - Percepio Tracealyzer. На момент выхода этой книги, Tracealyzer предоставляет самый широкий набор функций для анализа и мониторинга системы FreeRTOS. Он позволяет отслеживать производительность системы в реальном времени, анализировать данные с помощью графиков и диаграмм, искать проблемы и оптимизировать работу системы.

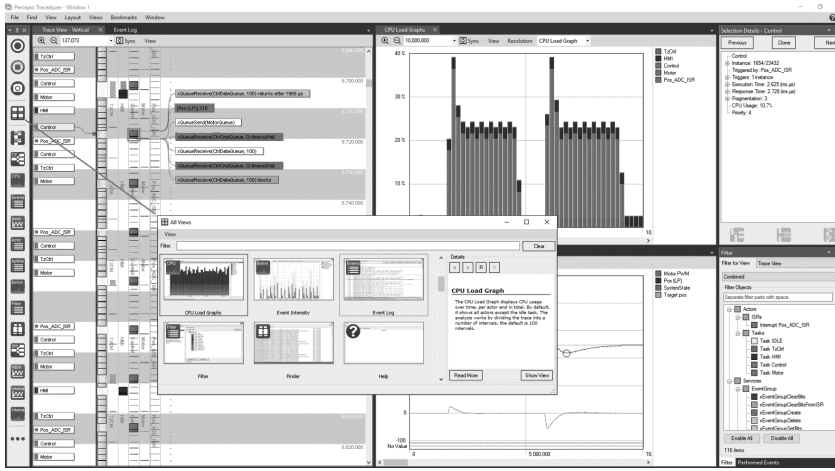


Рисунок 76. Окно программы Perceptio Tracealyzer.

По своей сути Tracealyzer является хорошо проработанной библиотекой при помощи макросов дополняющей исходный код операционной системы средствами сбора статистической информации, а также функциями передачи этой статистической информации на персональный компьютер по отладочному интерфейсу.

Из этого следует, что для того, чтобы приступить к мониторингу, проект предстоит немного модифицировать. Точнее, интегрировать в него дополнительный функционал. Это делается путем включения в проект дополнительных файлов.

```
/* Integrates the Tracealyzer recorder with FreeRTOS */
#if ( configUSE_TRACE_FACILITY == 1 )
        #include "trcRecorder.h"
#endif
```

```
myBoardInit();
...
/* Init only, trace starts later...*/
vTraceEnable(TRC_INIT);
...
/* RTOS scheduler starts */
vTaskStartScheduler();
...
```

```
/* In a task or ISR */
vTraceEnable(TRC_START);
```

Небольшие изменения для инициализации функционала

```
traceString adc_uechannel = xTraceRegisterString("ADC User Events");
...
vTracePrintF(adc_uechannel, "ADC channel %d: %d volts",
             ch, adc_reading);
```

Как вы можете видеть, в том, что делает Traceolyzer, нет ничего необычного. Код, встраиваемый в ядро операционной системы, эффективно выводит информацию о процессе функционирования операционной системы, а специализированное приложение на ПК анализирует и визуализирует полученную информацию.

Таким образом, пользуясь этой подсказкой вы можете реализовать собственную библиотеку – надстройку над FreeRTOS для мониторинга и оценки производительности.

Средства операционной системы

Это достаточно обширный раздел. С другой стороны, это достаточно сумбурный раздел. Мы уже не увидим чего-то принципиально нового в операционной системе и ее инструментарии. Остаются только небольшие, приятные функции, которые мы можем использовать для того, чтобы узнать о процессе выполнения кода немного больше.

Речь пойдет о встроенном функционале операционной системы, предназначенном для сбора статистической и отладочной информации. Как и подавляющее большинство вспомогательных функций, эти инструменты должны быть активированы за счет включения определенных констант в конфигурационный файл FreeRTOSConfig.h. Для того, чтобы при компиляции в состав проекта были включены функции сбора и форматирования статистической информации необходимо определить следующие константы:

```
#define configGENERATE_RUN_TIME_STATS            1
#define configUSE_STATS_FORMATTING_FUNCTIONS     1
```

Статистика времени выполнения задачи

Статистика времени выполнения (англ. Run-time) задачи предоставляет информацию о количестве времени, полученного каждой задачей. Время выполнения задачи — это общее время, в течение которого задача находилась в состоянии «Выполняется» с момента загрузки приложения.

Уверен, вы уже догадались, что речь идет о публикации данных счетчика, находящегося в контрольном блоке каждой задачи.

```
typedef struct tskTaskControlBlock
{
        volatile StackType_t     *pxTopOfStack;
        ListItem_t               xStateListItem;
        ListItem_t               xEventListItem;
        UBaseType_t              uxPriority;
        StackType_t              *pxStack;
        char                     pcTaskName[ configMAX_TASK_NAME_LEN ];

        ...

        #if( configGENERATE_RUN_TIME_STATS == 1 )
                uint32_t ulRunTimeCounter;
        #endif

} tskTCB;
```

Накапливаемые во время выполнения задачи статистические данные содержатся в переменной ulRunTimeCounter. Они могут быть использованы в качестве средства профилирования и отладки на этапе разработки проекта. Следует заметить, что информация будет актуальна только до переполнения переменной. А второе - сбор статистики во время выполнения увеличивает время переключения контекста задачи.

Чтобы получить бинарную статистику времени выполнения, используйте функцию API uxTaskGetSystemState().

```
UBaseType_t uxTaskGetSystemState(TaskStatus_t * const pxTaskStatusArray,
                        const UBaseType_t uxArraySize,
                        unsigned long * const pulTotalRunTime );
```

Если кратко, то uxTaskGetSystemState() заполняет структуру TaskStatus_t для каждой задачи, созданной в системе. Структура

TaskStatus_t содержит, среди прочего, дескриптор задачи, имя, приоритет, состояние и общее количество затраченного времени выполнения.

```
typedef struct xTASK_STATUS
{
        TaskHandle_t xHandle;
        const char *pcTaskName;
        UBaseType_t xTaskNumber;
        eTaskState eCurrentState;
        UBaseType_t uxCurrentPriority;
        UBaseType_t uxBasePriority;
        uint32_t ulRunTimeCounter;
        uint16_t usStackHighWaterMark;
} TaskStatus_t;
```

В качестве возвращаемого значения мы получим количество структур TaskStatus_t, заполненных функцией uxTaskGetSystemState(). Это число совпадает со значением возвращаемым функцией uxTaskGetNumberOfTasks(), но будет равно нулю, если значение, переданное в параметре uxArraySize, было меньше количества задач.

Прошу вас, помните, эта функция предназначена только для отладки, так как ее использование приводит к приостановке работы планировщика на достаточно длительный период, необходимый для формирования всей статистической информации и ее копирования в возвращаемые структуры.

Если вас интересует статистика времени выполнения какой-либо одной задачи, то вместо uxTaskGetSystemState() будет удобнее использовать функцию vTaskGetTaskInfo().

```
void vTaskGetTaskInfo( TaskHandle_t xTask, TaskStatus_t *pxTaskStatus,
                       BaseType_t xGetFreeStackSpace,
                       eTaskState eState );
```

Как видите, vTaskGetTaskInfo() заполняет структуру TaskStatus_t для одной выбранной задачи. Структура TaskStatus_t содержит, среди прочего, дескриптор задачи, имя, приоритет, состояние и общее количество затраченного времени выполнения.

Предыдущие функции снабжают нас информацией в бинарном формате, но существует и возможность получения «развернутого»

текстового ответа. Вспомогательная функция vTaskGetRunTimeStats() предназначена для сбора и форматирования информации о времени выполнения. Результат работы этой функции очень похож на результат работы диспетчера задач операционной системы Windows.

Главной особенностью функция vTaskGetRunTimeStats() является способность форматного вывода статистики времени выполнения в виде удобочитаемой таблицы.

```
Task            Abs Time      % Time
****************************************


uIP             12050         <1%
IDLE            587724        24%
QProdB2         2172          <1%
QProdB3         10002         <1%
QProdB5         11504         <1%
QConsB6         11671         <1%
```

Рисунок 77. Пример информации о задачах.

Это же является и главным недостатком данной функции. Форматный вывод в текстовом виде требует использования весьма требовательных к ресурсам функций из семейства printf(), что самым губительным образом сказывается на результате. А результат – искажение статистической информации в связи с вносимыми функцией vTaskGetRunTimeStats() затратами.

Не думаю, что стоит рассматривать подобные функции подробно, но обратите внимание, как функции удалось вычислить нагрузку? И каким образом вы можете определить сколько времени получает каждая из выполняемых в системе задач?

Секрет достаточно прост. Для сбора статистики можно использовать любой доступный таймер, запущенный на частоте, превышающей частоту тиковых прерываний в системе. Чем больше, тем точнее вы можете измерить время, отведенное планировщиком каждой задаче.

```
extern volatile unsigned long ulHighFrequencyTimerTicks;
#define portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()
                                ( ulHighFrequencyTimerTicks = 0UL )
#define portGET_RUN_TIME_COUNTER_VALUE() ulHighFrequencyTimerTicks
```

Выше приведен пример определений для сбора статистики. В этом примере существует некоторый таймер, работающий на частоте 20 килогерц. Это больше частоты тиковых прерываний. Значение счетчика таймера находится в переменной ulHighFrequencyTimerTicks. Прерывание от таймера просто увеличивает значение счетчика на единицу. Также мы определяем два макроса. portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() – сбрасывает счетчик. portGET_RUN_TIME_COUNTER_VALUE() – позволяет узнать текущее значение счетчика. Именно эти макросы использует функция vTaskGetRunTimeStats для получения и сбора статистической информации.

```
void vAFunction( void )
{
   static char cBuffer[ BUFFER_SIZE ];
   vTaskGetRunTimeStats( cBuffer );
}
```

К этому же семейству нативных средств сбора статистической информации можно и отнести функцию форматирующую базовую статистическую информацию о существующих в системе задачах.

```
void vTaskList( signed char *pcWriteBuffer );
```

Для демонстрации работы этой функции создадим массив символов

```
static char cBuffer[ 255 ];
```

и выполним функцию vTaskList() в любой задаче

```
void StartTask04(void const * argument)
{
  for(;;)
  {
     vTaskList( cBuffer );
     osDelay(100);
  }
}
```

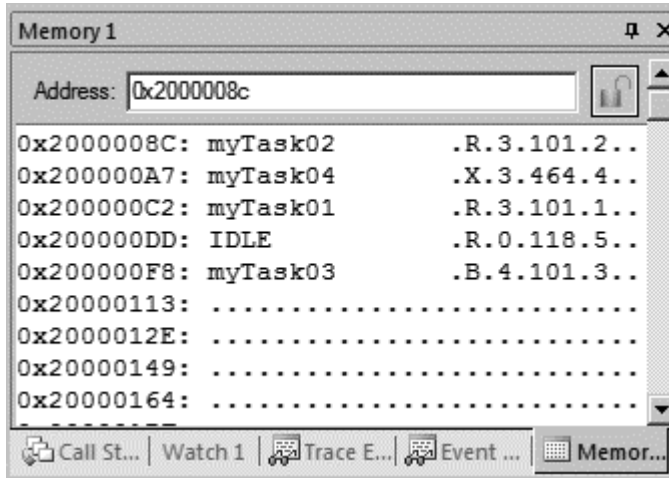


Рисунок 78. Пример визуализации работы функции vTaskList в среде разработки Keil uVision.

Величина стека

На страницах этой книги уже обсуждался вопрос стека, выделяемого для каждой задачи при ее создании. При этом каждый разработчик прекрасно знает сколько проблем возникает при исчерпании этого объёма.

Объем стека, используемого задачей, увеличивается и уменьшается по мере выполнения задачи и обработки прерываний. Функция uxTaskGetStackHighWaterMark() используется для того, чтобы определить насколько близка задача к переполнению выделенного ей пространства стека. Это значение называется «максимальной отметкой» стека. Это объем стека, который оставался неиспользованным, когда использование стека было максимальным (или самым глубоким). Чем ближе верхняя отметка к нулю, тем ближе задача подошла к переполнению своего стека.

```
UBaseType_t uxTaskGetStackHighWaterMark( TaskHandle_t xTask );
```

Как видно из прототипа, функция возвращает минимальный достигнутый в процессе работы размер стека для выбранной задачи.

```
void vTask1( void * pvParameters )
{
   UBaseType_t uxHighWaterMark;
```

```
    uxHighWaterMark = uxTaskGetStackHighWaterMark( NULL );

    for( ;; )
    {
        vTaskDelay( 1000 );
        uxHighWaterMark = uxTaskGetStackHighWaterMark( NULL );
    }
}
```

Для текущей задачи мы можем указать NULL в качестве дескриптора задачи. В общем виде это может выглядеть как показано на примере выше.

Функции обратного вызова

В предыдущих главах мы уже успели познакомиться с некоторыми перехватчиками, функциями обратного вызова (англ. Hook Function), которые ядро операционной системы вызывает при возникновении некоторых ситуаций.

К примеру, в главе, посвященной задачам мы говорили о функции обратного вызова, получающей управление каждый раз, когда ядро операционной системы собирается утилизировать оставшееся неиспользованным свободное процессорное время в задаче простоя.

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

Вы наверняка помните, что FreeRTOS будет использовать этот прототип только если в конфигурационном файле FreeRTOSConfig.h определить константу configUSE_IDLE_HOOK и присвоить ей значение 1.

Функция обратного вызова, воплощающая в себе все операции, связанные с простоем, - это хорошее место не только для перехода в состояние сна, но и для анализа реальной нагрузки на процессор. По наличию или отсутствую переходов к обработчику вы можете с уверенностью судить о том, не является ли нагрузка на процессор критичной.

Еще одной, достаточно часто используемой функцией обратного вызова является Tick Hook. Разрешить использование этого обратного вызова можно добавив в конфигурационный файл FreeRTOSConfig.h константу configUSE_TICK_HOOK и присвоив ей значение 1. После этого вы

можете определить в коде создаваемого проекта функцию обратного вызова:

```
void vApplicationTickHook( void );
```

Эта функция не только идеальное место для размещения функциональности счетчиков, но и возможность для трассировки, отслеживания тиковых прерываний.

Стоит только помнить, что даже небольшое увеличение времени выполнения часто повторяемых прерываний, как, например, тиковое прерывание системы, может привести к огромным потерям времени приложения. Относитесь к этому с должной осмотрительностью ведь vApplicationTickHook() выполняется из обработчика тикового прерывания, поэтому он должен быть очень коротким, не использовать большое количество стека и не вызывать никаких функций API, которые не заканчиваются на «FromISR» или «FROM_ISR».

С тем, что в данном разделе мы рассматриваем функции обратного вызова с точки зрения использования их функционала в целях отладки, то полезно будет упомянуть и о обработке отказов в выделении памяти. Как вы помните, схемы распределения памяти, реализованные в heap_1.c, heap_2.c, heap_3.c, heap_4.c и heap_5.c, могут дополнительно включать обработчик отказа malloc() (или функцию обратного вызова), который будет произведен, если pvPortMalloc() когда-либо вернет значение NULL.

Это достаточно полезная функция, которую можно использовать для диагностики проблем, связанных с нехваткой памяти. Это особенно актуально в том случае, если в вашем приложении вызов pvPortMalloc() происходит не в контексте операционной системы. Прототип функции имеет вид:

```
void vApplicationMallocFailedHook( void );
```

А сама функция будет вызвана только в том случае, если в конфигурационном файле FreeRTOSConfig.h вы определите константу configUSE_MALLOC_FAILED_HOOK и присвоите ей значение 1.

Последняя функция обратного вызова, о которой стоит упомянуть в этом разделе, - это запуск задачи демона (англ. Daemon Task Startup). Мы

уже знакомы с этим демоном, в частности, именно его мы рассматривали как контекст для выполнения функционала программных таймеров.

В большинстве статей функция обратного вызова, вызываемая при старте задачи демона, рекомендуется к использованию в качестве загрузчика конфигурации. Да, действительно, запуск демона произойдет в то время, когда все задачи и коммуникационные элементы ядра будут уже созданы. С другой стороны, использование столь низкоприоритетной задачи для загрузки конфигурации вызывает много споров.

Прототип функции обратного вызова имеет следующий вид:

```
void vApplicationDaemonTaskStartupHook( void );
```

Использование функции станет возможным только после того, как в конфигурационном файле FreeRTOSConfig.h константе configUSE_DAEMON_TASK_STARTUP_HOOK будет установлено значение 1.

Переполнение стека

Не стоит тратить время на описание последствий переполнения стека. В случае с операционной системой эта проблема осложняется тем фактом, что мы поддерживаем отдельный стек для каждой создаваемой задачи.

FreeRTOS предоставляет встроенный механизм контроля, который можно использовать для помощи по обнаружению и исправлению случаев переполнения стека. Для управления используется параметр – константа configCHECK_FOR_STACK_OVERFLOW определяемая в конфигурационном файле FreeRTOSConfig.h.

Прототип функции обратного вызова имеет вид:

```
void vApplicationStackOverflowHook( TaskHandle_t xTask,
                                    signed char *pcTaskName );
```

Обратите внимание, что использование контроля переполнения стека доступно только для архитектур, в которых память не сегментирована. Кроме того, некоторые архитектуры умеют генерировать ошибку или исключение в ответ на повреждение стека до того, как эта проверка будет проведена ядром RTOS.

Параметры xTask и pcTaskName передают функции обратного вызова дескриптор и имя задачи в которой произошло переполнение стека. Стоит учитывать, что при масштабных повреждениях стека эти параметры также могут быть повреждены. В этом случае у вас останется только шанс использовать переменную pxCurrentTCB напрямую. Следует помнить, что дополнительная проверка на переполнение стека серьезно влияет на производительность системы. Разработчики FreeRTOS рекомендуют включать данный функционал только на время тестирования и отладки приложения.

Метод 1. Вполне вероятно, что стек достигнет своего наибольшего (самого глубокого) значения после того, как ядро RTOS выведет задачу из состояния «Выполняется», потому что именно тогда стек будет содержать контекст задачи. В этот момент ядро RTOS может проверить, что указатель стека процессора остается в допустимом пространстве стека. И если при использовании данного метода произошел вызов функции vApplicationStackOverflowHook, это однозначно показывает, что указатель стека задачи выходит за допустимый диапазон адресов стека. Метод достаточно быстрый, но не гарантирует, что будут обранужены все переполнения стека. Для того, чтобы использовать данный метод, константа configCHECK_FOR_STACK_OVERFLOW должна иметь значение 1.

Метод 2. При создании задачи ее стек заполняется известным значением (окрашивание стека). При переходе задачи из состояния «Выполняется» ядро RTOS проверяет последние 16 байтов в допустимом диапазоне адресов стека, чтобы убедиться, что эти известные значения не были перезаписаны задачей или действием прерывания. Если начальное значение этих ячеек памяти будет перезаписано другим значением, то произойдет вызов vApplicationStackOverflowHook. Этот метод менее эффективен, но все же довольно быстр. Вероятность обнаружения переполнения стека весьма высока. Для использования этого метода установите параметр configCHECK_FOR_STACK_OVERFLOW в значение 2.

Глава 14. Макросы

Если обратиться к оригинальной документации на FreeRTOS, то вы найдете в ней целый раздел, посвященный так называемым «макросам – ловушкам трассировки». По своей сути это очень хорошая идея. Ключевые моменты исходного кода операционной системы, моменты, представляющие интерес с точки зрения отладки и отслеживания поведения создаваемого приложения, содержат вызовы макросов. По умолчанию эти макросы пустые и никак не влияют на исходный код.

Однако, программист в любой момент может переопределить эти макросы, включив в них свой собственный функционал. Это можно сделать как с целью расширения средств трассировки, так и для привнесения собственного функционала в код операционной системы. Приложение должно переопределять только те макросы, которые представляют особый интерес. Неиспользуемые макросы остаются пустыми и, следовательно, не влияют на размер кода и ход выполнения приложения.

Это очень удобный прием, позволяющий нам модифицировать исходный код, не затрагивая его исходные файлы. А самое большое преимущество такого подхода проявится, когда мы решим обновить исходные файлы операционной системы, что можно сделать без какого-либо влияния на прикладную часть.

Давайте посмотрим на то, как авторы реализовали механизм макросов. Возьмем функцию с минимальным количеством кода:

```
void vQueueDelete ( QueueHandle_t xQueue )
{
   Queue_t * const pxQueue = xQueue;
   configASSERT( pxQueue );
   traceQUEUE_DELETE( pxQueue );
   vQueueUnregisterQueue( pxQueue );
   if( pxQueue->ucStaticallyAllocated == ( uint8_t ) pdFALSE )
   {
      vPortFree( pxQueue );
   }
   else
   {
      mtCOVERAGE_TEST_MARKER();
   }
}
```

В теле функции вы можете увидеть вызов макроса – ловушки трассировки traceQUEUE_DELETE(pxQueue). Он начинается с префикса trace. Как и все макросы он объявлен в файле FreeRTOS.h:

```
#ifndef traceQUEUE_DELETE
        #define traceQUEUE_DELETE( pxQueue )
#endif
```

Именно механизм макросов – ловушек был использован автором книги для иллюстрации процесса переключения задач и визуализации основных временных процессов в данной книге. Для этого, в файле FreeRTOSConfig.h были переопределены два макроса вызывающиеся при входе и выходе из задачи. Макросы теперь ссылаются на функции, а в качестве параметров в функции передаются метки, содержащиеся в контрольном блоке задачи (TCB) и идентифицирующие их.

```
#define traceTASK_SWITCHED_IN() TaskSwitchedIn((int)pxCurrentTCB->pxTaskTag);
#define traceTASK_SWITCHED_OUT() TaskSwitchedOut((int)pxCurrentTCB->pxTaskTag);
```

Для полноты примера рассмотрим на то, как выглядят функции TaskSwitchedIn() и TaskSwitchedOut().

```
void TaskSwitchedIn(int tag)
{
  switch (tag)
  {
     case 1:
       GPIOA->BSRR = GPIO_PIN_0;
       break;
     case 2:
       GPIOA->BSRR = GPIO_PIN_1;
       break;
     case 3:
       GPIOA->BSRR = GPIO_PIN_2;
       break;
     case 4:
       GPIOA->BSRR = GPIO_PIN_3;
       break;
  }
}

void TaskSwitchedOut(int tag)
{
  switch (tag)
  {
```

```
      case 1:
        GPIOA->BSRR = (uint32_t)GPIO_PIN_0 << 16U;
        break;
      case 2:
        GPIOA->BSRR = (uint32_t)GPIO_PIN_1 << 16U;
        break;
      case 3:
        GPIOA->BSRR = (uint32_t)GPIO_PIN_2 << 16U;
        break;
      case 4:
        GPIOA->BSRR = (uint32_t)GPIO_PIN_3 << 16U;
        break;
  }
}
```

Таким образом, удалось реализовать переключение состояния портов ввода-вывода в зависимости от того, задача с какой меткой в настоящее время получает управление.

Макросы, вне всякого сомнения, являются инструментом удобным, но существует несколько моментов, о которых необходимо помнить и учитывать при написании собственного кода:

1. Если вы используете макросы, которые вызываются из обработчиков прерываний, а в особенности из обработчика тиковых прерываний, то такие макросы должны выполняться максимально быстро и не занимать много места в стеке.
2. Определения макросов должно происходить до включения заголовочного файла FreeRTOS.h. Проще всего определять макросы – ловушки в нижней части конфигурационного файла FreeRTOSConfig.h или в отдельном заголовочном файле, включенном в конец FreeRTOSConfig.h.

Нет какого-то универсального сценария использования макросов отладки и трассировки. Ознакомившись с тем, в каких участках кода происходит вызов каждого из них, вы наверняка придумаете им достойное применение. Дальнейшие разделы содержат краткую информацию о предназначении определенных макросов – ловушек и их вызовах из различных функций API операционной системы.

traceINCREASE_TICK_COUNT(xTicksToJump) - Вызывается из vTaskStepTick() после того, как произойдет увеличение счетчика тиков.

traceLOW_POWER_IDLE_BEGIN() – Вызывается из portTASK_FUNCTION() до начала простоя.

traceLOW_POWER_IDLE_END() – Вызывается из portTASK_FUNCTION() после пробуждения.

traceMOVED_TASK_TO_READY_STATE(xTask) – Макрос вызывается, когда задача переходит в состояние готовности.

tracePOST_MOVED_TASK_TO_READY_STATE(pxTCB) – Вызывается из макроса prvAddTaskToReadyList() после того, как задача будет успешно перемещена в конец списка задач, находящихся в состоянии готовности.

traceTASK_CREATE(xTask) – Вызывается из функций xTaskCreate() и xTaskCreateStatic() когда задача успешно создана.

traceTASK_CREATE_FAILED(pxNewTCB) – Вызывается из функций xTaskCreate() и xTaskCreateStatic(), если в процессе создания задачи возникла ошибка, связанная с выделением памяти из кучи; как правило, в куче не хватает памяти.

traceTASK_DELAY() – Вызывается из функции vTaskDelay().

traceTASK_DELAY_UNTIL() – Вызывается из функции vTaskDelayUntil().

traceTASK_DELETE(xTask) – Вызывается из функции vTaskDelete().

traceTASK_INCREMENT_TICK(xTickCount) – Вызывается в ходе тикового прерывания.

traceTASK_NOTIFY(uxIndexToNotify) - Вызывается из функции xTaskGenericNotify(), когда происходит уведомление задачи о значении уведомления.

traceTASK_NOTIFY_FROM_ISR(uxIndexToNotify) – Вызывается из функции xTaskGenericNotifyFromISR() до того, как из обработчика прерывания будет отправлено значение уведомления задаче.

traceTASK_NOTIFY_GIVE_FROM_ISR(uxIndexToNotify) - Вызывается из функции vTaskGenericNotifyGiveFromISR() до того, как из обработчика прерывания значение уведомления задачи будет увеличено.

traceTASK_NOTIFY_TAKE(uxIndexToWait) – Вызывается из функции ulTaskGenericNotifyTake() при чтении и уменьшении значения уведомления.

traceTASK_NOTIFY_TAKE_BLOCK(uxIndexToWait) – Вызывается из функции ulTaskGenericNotifyTake(), когда задача заблокирована из-за того, что значение уведомления стало нулевым.

traceTASK_NOTIFY_WAIT(uxIndexToWait) – Вызывается из xTaskGenericNotifyWait(), когда происходит чтение и возврат значения уведомления. Значение не изменяется.

traceTASK_NOTIFY_WAIT_BLOCK(uxIndexToWait) – Вызывается из функции xTaskGenericNotifyWait(), когда задача заблокирована и ожидает уведомления.

traceTASK_PRIORITY_DISINHERIT(pxTCBOfMutexHolder, uxOriginalPriority) – Вызывается из функции xTaskPriorityDisinherit() перед тем, как произойдет понижение приоритета задачи, которая раньше разблокировала мьютекс.

traceTASK_PRIORITY_INHERIT(pxTCBOfMutexHolder, uxInheritedPriority) – Вызывается из функции xTaskPriorityInherit() после того, как произошло повышение приоритета задачи держателе мьютекса, который ожидает задача с более высоким приоритетом.

traceTASK_PRIORITY_SET(xTask,uxNewPriority) - Вызывается из функции vTaskPrioritySet().

traceTASK_RESUME(xTask) – Вызывается из функции vTaskResume().

traceTASK_RESUME_FROM_ISR(xTask) – Вызывается из функции xTaskResumeFromISR().

traceTASK_SUSPEND(xTask) – Вызывается из функции vTaskSuspend().

traceTASK_SWITCHED_IN() – Вызывается после того, как планировщик выберет задачу для запуска. В этот момент pxCurrentTCB содержит дескриптор задачи, которая вот-вот перейдет в состояние выполняется.

traceTASK_SWITCHED_OUT() – Макрос вызывается перед тем, как планировщик выберет новую задачу для выполнения. В этот момент pxCurrentTCB содержит дескриптор задачи, которая будет выведена из состояния выполнения.

Очереди

traceBLOCKING_ON_QUEUE_PEEK(pxQueue) - Вызывается из xQueuePeek() перед тем, как задача, пытающаяся просмотреть пустую очередь, перейдет в состояние блокировки

traceBLOCKING_ON_QUEUE_RECEIVE(xQueue) – Вызов этого макроса указывает на тот факт, что выполняемая в данный момент задача будет заблокирована в результате попытки чтения из пустой очереди или при попытке «взять» пустой семафор или мьютекс.

traceBLOCKING_ON_QUEUE_SEND(xQueue) – Вызов макроса указывает на тот факт, что выполняемая в данный момент задача будет заблокирована после попытки записи в полную очередь.

traceCREATE_COUNTING_SEMAPHORE() – Вызывается из функции xSemaphoreCreateCounting(),если семафор был успешно создан.

traceCREATE_COUNTING_SEMAPHORE_FAILED() – Вызывается из функции xSemaphoreCreateCounting() в том случае, если семафор не может быть создан из-за недостатка памяти.

traceCREATE_MUTEX(pxNewMutex) - Вызывается из функции xSemaphoreCreateMutex(), если мьютекс успешно создан.

traceCREATE_MUTEX_FAILED() – Вызывается из функции xSemaphoreCreateMutex(), если при создании мьютекса произошла ошибка связанная с выделением памяти в куче.

traceGIVE_MUTEX_RECURSIVE(xMutex) – Вызывается из функции xSemaphoreGiveRecursive(), если мьютекс успешно «отдан».

traceGIVE_MUTEX_RECURSIVE_FAILED(xMutex) – Вызывается из функции xSemaphoreGiveRecursive(), если мьютекс не был успешно предоставлен в связи с тем, что вызывающая задача не была владельцем (держателем) мьютекса.

traceQUEUE_CREATE(pxNewQueue) – Вызывается из функций xQueueCreate() и xQueueCreateStatic(), если очередь была успешно создана.

traceQUEUE_CREATE_FAILED() – Вызывается из функций xQueueCreate() и xQueueCreateStatic(), если создать очередь требуемого размера не удалось в связи с тем, что невозможно выделить требуемый объём памяти из кучи.

traceQUEUE_DELETE(xQueue) – Вызывается из функции vQueueDelete().

traceQUEUE_PEEK(xQueue) – Вызывается из функции xQueuePeek()

traceQUEUE_PEEK_FAILED(pxQueue) - Вызывается из функции xQueuePeek(), когда очередь пуста даже после истечения времени таймаута или если таймаут не определен.

traceQUEUE_PEEK_FROM_ISR(pxQueue) – Вызывается из функции xQueuePeekFromISR() перед извлечением элемента из очереди.

traceQUEUE_PEEK_FROM_ISR_FAILED(pxQueue) – Будет вызван из функции xQueuePeekFromISR(), если очередь пустая.

traceQUEUE_REGISTRY_ADD(xQueue, pcQueueName) – Вызывается из vQueueAddToRegistry() после успешного добавления очереди в реестр.

traceQUEUE_RECEIVE(xQueue) – Вызывается из функции xQueueReceive() либо из любой функции «взятия» семафора, когда получение из очереди прошло успешно.

traceQUEUE_RECEIVE_FAILED(xQueue) – Вызывается из функции xQueueReceive() либо любой другой функции «взятия» семафора, когда операция получения из очереди закончилась неудачей из-за того, что очередь была пуста.

traceQUEUE_RECEIVE_FROM_ISR(xQueue) – Вызывается из функции xQueueReceiveFromISR(), когда операция получения завершилась успешно.

traceQUEUE_RECEIVE_FROM_ISR_FAILED(xQueue) – Вызывается из xQueueReceiveFromISR() в случае, если операция получения завершилась сбоем по причине того, что очередь уже пуста.

traceQUEUE_SEND(xQueue) - Вызывается из функций xQueueSend(), xQueueSendToFront(), xQueueSendToBack() или любой другой функции «дающей» семафор, когда отправка в очередь закончилась успешно.

traceQUEUE_SEND_FAILED(xQueue) - Вызывается из функций xQueueSend(), xQueueSendToFront(), xQueueSendToBack() или любой другой функции «дающей» семафор» когда операция отправки в очередь завершилась неудачей из-за переполнения очереди.

traceQUEUE_SEND_FROM_ISR(xQueue) – Вызывается из xQueueSendFromISR(), когда операция отправки выполнена успешно.

traceQUEUE_SEND_FROM_ISR_FAILED(xQueue) – Вызывается из xQueueSendFromISR() при сбое операции отправки, вызванном тем, что очередь уже полностью заполнена.

traceTAKE_MUTEX_RECURSIVE(xMutex) – Вызывается из функции xQueueTakeMutexRecursive().

traceTAKE_MUTEX_RECURSIVE_FAILED(xMutex) – Вызывается из xQueueTakeMutexRecursive(), когда вызывающая задача не удерживает рекурсивный мьютекс и не может его «отдать».

Таймера

tracePEND_FUNC_CALL(xFunctionToPend, pvParameter1, ulParameter2, ret) – Данный макрос вызывается из функции xTimerPendFunctionCall() после отправки ожидающей вызова функции в очередь.

tracePEND_FUNC_CALL_FROM_ISR(xFunctionToPend, pvParameter1, ulParameter2, ret) – Вызывается из xTimerPendFunctionCallFromISR() после отправки ожидающей вызова функции в очередь.

traceTIMER_COMMAND_RECEIVED(pxTimer, xCommandID, xCommandValue) – Данный макрос вызывается службой таймера каждый раз, когда служба получает команду прежде чем команда будет фактически обработана.

traceTIMER_COMMAND_SEND(pxTimer, xCommandID, xOptionalValue, xStatus) – Макрос вызывается из любой функции API, которая отправляет команды задаче обслуживания программных таймеров, например, xTimerReset(), xTimerStop() и подобные. Параметр xStatus будет иметь значение pdFAIL, если команду не удалось отправить в очередь команд таймера.

traceTIMER_CREATE(pxNewTimer) - Макрос вызывается из xTimerCreate(), если программный таймер был успешно создан.

traceTIMER_CREATE_FAILED() – Вызывается из xTimerCreate(), если создание таймера закончилось неудачей из-за того, что в куче недостаточно свободного места.

traceTIMER_EXPIRED(pxTimer) – Макрос вызывается службой программных таймеров после истечения периода программного таймера, до того, как будет вызвана функция таймера.

Группы событий

traceEVENT_GROUP_CLEAR_BITS(xEventGroup, uxBitsToClear) – Макрос вызывается из функции xEventGroupClearBits() до того, как выбранные биты будут очищены и возвращены к предыдущему значению.

traceEVENT_GROUP_CLEAR_BITS_FROM_ISR(xEventGroup, uxBitsToClear) – Вызывается изфункции xEventGroupClearBitsFromISR() перед вызовом xEventGroupClearBits().

traceEVENT_GROUP_CREATE(xEventGroup) – Вызывается из xEventGroupCreate() при успешном создании группы событий.

traceEVENT_GROUP_CREATE_FAILED() – Вызывается из функции xEventGroupCreate(), если при создании группы событий возникла ошибка.

traceEVENT_GROUP_DELETE(xEventGroup) – Вызывается из функции vEventGroupDelete() перед тем, как группа событий будет удалена.

traceEVENT_GROUP_SET_BITS(xEventGroup, uxBitsToSet) – Вызывается из xEventGroupSetBits() перед установкой выбранных битов событий и потенциальной разблокировкой задач.

traceEVENT_GROUP_SET_BITS_FROM_ISR(xEventGroup, uxBitsToSet) – Вызывается из xEventGroupSetBitsFromISR() перед вызовом xEventGroupSetBits().

traceEVENT_GROUP_SYNC_BLOCK(xEventGroup, uxBitsToSet, uxBitsToWaitFor) – Вызывается из xEventGroupSync() перед блокировкой.

traceEVENT_GROUP_WAIT_BITS_BLOCK(xEventGroup, uxBitsToWaitFor) – Вызывается из xEventGroupWaitBits() перед блокировкой для ожидания установки необходимых битов событий.

Куча

traceFREE(pvAddress, uiSize) – Вызывается из функции порта vPortFree() при освобождении памяти.

traceMALLOC(pvAddress, uiSize) – Вызывается из функции порта pvPortMalloc() при выделении памяти.

Заключение

Рассказ об операционной системе FreeRTOS никогда не будет полным и исчерпывающим. Проект жив, он показал свою жизнеспособность и востребованность не только тем, что за годы своего существования де факто стал промышленным стандартом, но и по той, причине, что продолжает развиваться вместе с современными микроконтроллерами. Проект меняется, что можно отследить, в частности, читая данную книгу. Первые заметки были написаны в момент перехода к FreeRTOS V10.2.1, а последние главы V10.5.1.

Как показывает опыт, именно FreeRTOS является лучшим примером для изучения операционных систем созданных для микроконтроллеров.

Об авторе

Владимир В. Мединцев

Больше 25 лет проработал в IT сфере. Принимал участие в реализации крупнейших нефтегазовых и энергетических проектов современной России: КТК, Голубой поток, Сахалин-2, Северный поток, принимал участие в реконструкции Хабаровского НПЗ, строительстве электростанций в Астрахани и Буденновске, реконструкции защитного укрытия Чернобыльской АЭС.

Автор и руководитель учебного курса «Инженер умных устройств» для образовательной платформы «GeekBrains». Автор учебных курсов «Операционная система FreeRTOS», «Вводный курс. Микроконтроллеры на ядре ARM», «Проектирование устройств на микроконтроллерах». Ведет собственный YouTube канал о электронике и программировании.