Продолжение. Начало в № 2`2011

FreeRTOS — операционная система для микроконтроллеров

Андрей КУРНИЦ
kurnits@stim.by

В этой статье будет продолжено изучение FreeRTOS — операционной системы для микроконтроллеров. Здесь описан процесс принудительного изменения приоритета задач в ходе их выполнения, показано, как динамически создавать и уничтожать задачи. Рассмотрен вопрос о том, как расходуется память при создании задач. Подведен итог по вытесняющей многозадачности во FreeRTOS и рассказано о стратегии назначения приоритетов задачам под названием Rate Monotonic Scheduling. Далее мы обсудим тему кооперативной многозадачности, ее преимущества и недостатки и приведем пример программы, использующей кооперативную многозадачность во FreeRTOS. Автор уделил внимание и альтернативным схемам планирования: гибридной многозадачности и вытесняющей многозадачности без разделения времени.

Динамическое изменение приоритета

При создании задачи ей назначается определенный приоритет. Однако во FreeRTOS есть возможность динамически изменять приоритет уже созданной задачи, даже после запуска планировщика. Для динамического изменения приоритета задачи служит API-функция ***vTaskPrioritySet()***. Ее прототип:

```
void vTaskPrioritySet( xTaskHandle pxTask, unsigned portBASE_
TYPE uxNewPriority );
```

Назначение параметров:

1. ***pxTask*** — дескриптор (handle) задачи, приоритет которой необходимо изменить. Дескриптор задачи может быть получен при создании экземпляра задачи API-функцией ***xTaskCreate()*** (параметр ***pxCreatedTask*** [1, № 3]). Если необходимо изменить приоритет задачи, которая вызывает API-функцию ***vTaskPrioritySet()***, то в качестве параметра ***pxTask*** следует задать NULL.
2. ***uxNewPriority*** — новое значение приоритета, который будет присвоен задаче. При задании приоритета больше (***configMAX_PRIORITIES*** — 1) приоритет будет установлен равным (***configMAX_PRIORITIES*** — 1).

Прежде чем изменить приоритет какой-либо задачи, может оказаться полезной возможность предварительно получить значение ее приоритета. API-функция ***uxTaskPriorityGet()*** позволяет это сделать. Ее прототип:

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

Назначение параметров и возвращаемое значение:

1. ***pxTask*** — дескриптор задачи, приоритет которой необходимо получить. Если необходимо получить приоритет задачи, которая вызывает API-функцию ***uxTaskPriorityGet()***, то в качестве параметра ***pxTask*** следует задать NULL.
2. Возвращаемое значение — непосредственно значение приоритета.

Наглядно продемонстрировать использование API-функций ***vTaskPrioritySet()*** и ***uxTaskPriorityGet()*** позволяет учебная программа № 1:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "FreeRTOS.h"
#include "task.h"

/* Прототипы функций, которые реализуют задачи. */
void vTask1( void *pvParameters );
void vTask2( void *pvParameters );

/* Глобальная переменная для хранения приоритета Задачи 2 */
xTaskHandle xTask2Handle;

/*-----------------------------------------------------------*/

int main( void )
{
   /* Создать Задачу 1, присвоив ей приоритет 2.
   Передача параметра в задачу, как и получение дескриптора
   задачи, не используется */
   xTaskCreate( vTask1, "Task 1", 1000, NULL, 2, NULL );

   /* Создать Задачу 2 с приоритетом = 1, меньшим,
   чем у Задачи 1. Передача параметра не используется.
   Получить дескриптор создаваемой задачи в переменную
   xTask2Handle */
   xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, &xTask2Handle );

   /* Запустить планировщик. Задачи начнут выполняться.
   Причем первой будет выполнена Задача 1 */
   vTaskStartScheduler();

   return 0;
}
/*-----------------------------------------------------------*/
```

```
/* Функция Задачи 1 */
void vTask1( void *pvParameters )
{
   unsigned portBASE_TYPE uxPriority;

   /* Получить приоритет Задачи 1. Он равен 2 и не изменяется
   на протяжении всего времени
   работы учебной программы № 1 */
   uxPriority = uxTaskPriorityGet( NULL );

   for( ;; )
   {
      /* Сигнализировать о выполнении Задачи 1 */
      puts("Task1 is running" );

      /* Сделать приоритет Задачи 2 на единицу больше
      приоритета Задачи 1 (равным 3).
      Получить доступ к Задаче 2 из тела Задачи 1 позволяет
      дескриптор Задачи 2, который сохранен в глобальной
      переменной xTask2Handle*/
      puts( "To raise the Task2 priority" );
      vTaskPrioritySet( xTask2Handle, ( uxPriority + 1 ) );

      /* Теперь приоритет Задачи 2 выше. Задача 1
      продолжит свое выполнениение лишь тогда,
      когда приоритет Задачи 1 будет уменьшен. */
   }
   vTaskDelete( NULL );
}

/*-----------------------------------------------------------*/
/* Функция Задачи 2 */
void vTask2( void *pvParameters )
{
unsigned portBASE_TYPE uxPriority;

   /* Получить приоритет Задачи 2. Так как после старта
   планировщика Задача 1 имеет более высокий приоритет,
   то если Задача 2 получает управление, значит, ее приоритет
   был повышен до 3 */
   uxPriority = uxTaskPriorityGet( NULL );

   for( ;; )
   {
      /* Сигнализировать о выполнении Задачи 2 */
      puts( "Task2 is running" );

      /* Задача 2 понижает свой приоритет на 2 единицы
      (становится равен 1). Таким образом, он становится ниже
      приоритета Задачи 1, и Задача 1 получает управление */
      puts( "To lower the Task2 priority" );
      vTaskPrioritySet( NULL, ( uxPriority - 2 ) );
   }
   vTaskDelete( NULL );
}
/*-----------------------------------------------------------*/
```

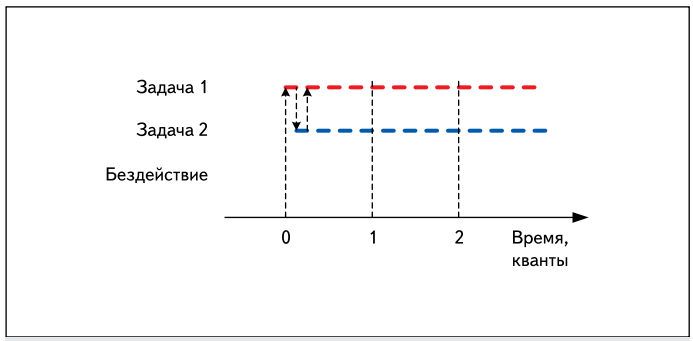

Рис. 1. Разделение процессорного времени между задачами в учебной программе № 1

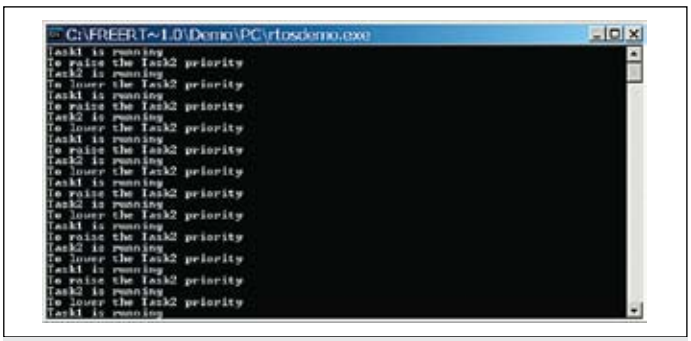

Рис. 2. Результат выполнения учебной программы № 1

В программе создана Задача 1 с приоритетом 2 и Задача 2 с приоритетом 1. Задача 1 повышает приоритет Задачи 2 так, чтобы он превысил приоритет Задачи 1. Задача 2, в свою очередь, понижает свой же приоритет так, чтобы он стал ниже приоритета Задачи 1. Задача 1, как и Задача 2, не переходит в блокированное состояние. Обе задачи поочередно получают процессорное время за счет периодического изменения приоритета Задачи 2 (он становится то ниже, то выше приоритета Задачи 1).

В момент запуска планировщика обе задачи готовы к выполнению (в учебной программе № 1 они вообще не переходят в блокированное состояние, то есть либо выполняются, либо находятся в состоянии готовности к выполнению). Управление получает Задача 1, так как ее приоритет (равен 2) больше, чем приоритет Задачи 2. После сигнализации о своей работе она вызывает API-функцию ***vTaskPrioritySet()***, вследствие чего Задача 2 получает приоритет выше, чем Задача 1 (он становится равным 3).

Вызов ***vTaskPrioritySet()*** помимо изменения приоритета приводит к тому, что управление получает планировщик, который запускает Задачу 2, так как приоритет у нее теперь выше.

Получив управление, Задача 2 сигнализирует о своем выполнении. После чего она понижает свой приоритет на две единицы, так, чтобы он стал меньше приоритета Задачи 1 (стал равен 1). После этого управление снова получает планировщик и так далее. Разделение процессорного времени в учебной программе № 1 показано на рис. 1, а результат ее выполнения — на рис. 2.

Следует отметить, что задачи сменяют друг друга с частотой, превышающей частоту системных квантов. Частота их следования зависит от быстродействия рабочей станции, и сообщения выводятся на экран с очень большой скоростью, поэтому для того, чтобы увидеть изображение на дисплее, соответствующее рис. 2, необходимо искусственно приостановить выполнение учебной программы.

Уничтожение задач

Задача может уничтожить саму себя или любую другую задачу в программе с помощью API-функции ***vTaskDelete()***. Удаленная задача физически не существует и, следовательно, никогда не выполняется. Нет возможности восстановить удаленную задачу, единственный выход — создать новую.

Ядро FreeRTOS устроено так, что внутренняя реализация задачи Бездействие отвечает за освобождение памяти, которую использовала удаленная задача. К программам, в которых происходит создание и удаление задач, предъявляется следующее требование. Если разработчик использует функцию-ловушку задачи Бездействие [1, № 4], то время выполнения этой функции должно быть меньше времени выполнения задачи Бездействие (то есть времени, пока нет ни одной задачи, готовой к выполнению).

Следует отметить, что при уничтожении задачи ядро освобождает лишь системную, не доступную прикладному программисту память, связанную с задачей. Вся память и другие ресурсы, которые программист использовал в задаче, также явно должны быть освобождены.

Прототип API-функции ***vTaskDelete()***:

```
void vTaskDelete( xTaskHandle pxTaskToDelete );
```

Единственный параметр ***pxTaskToDelete*** — это дескриптор задачи, которую необходимо уничтожить. Если необходимо уничтожить задачу, которая вызывает API-функцию ***vTaskDelete()***, то в качестве параметра ***pxTaskToDelete*** следует задать NULL.

Учебная программа № 2 демонстрирует динамическое создание и уничтожение задач:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "FreeRTOS.h"
#include "task.h"

/* Прототипы функций, которые реализуют задачи. */
void vTask1( void *pvParameters );
void vTask2( void *pvParameters );

/*-----------------------------------------------------------*/

int main( void )
{
   /* Статическое создание Задачи 1 с приоритетом 1 */
   xTaskCreate( vTask1, "Task 1", 1000, NULL, 1, NULL );
   /* Запустить планировщик. Задача 1 начнет выполняться */
   vTaskStartScheduler();
   return 0;
}
/*-----------------------------------------------------------*/
/* Функция Задачи 1 */
void vTask1( void *pvParameters )
{

   for( ;; )
   {
      /* Сигнализировать о выполнении Задачи 1 */
      puts("Task1 is running" );

      /* Динамически (после старта планировщика) создать
      Задачу 2 с приоритетом 2.
      Она сразу же получит управление */
      xTaskCreate( vTask2, "Task 2", 1000, NULL, 2, NULL );

      /* Пока выполняется Задача 2 с более высоким
      приоритетом, Задача 1 не получает процессорного
      времени. Когда Задача 2 уничтожила сама себя,
      управление снова получает Задача 1 и переходит
      в блокированное состояние на 100 мс. Так что в системе
      не остается задач, готовых к выполнению, и выполняется
      задача Бездействие */
      vTaskDelay( 100 );
   }
   vTaskDelete( NULL );
}

/*-----------------------------------------------------------*/
/* Функция Задачи 2 */
void vTask2( void *pvParameters )
{
   /* Задача 2 не делает ничего, кроме сигнализации о своем
   выполнении, и сама себя уничтожает. Тело функции
   не содержит бесконечного цикла, так как в нем нет
   необходимости. Тело функции Задачи 2 выполнится 1 раз,
   после чего задача будет уничтожена. */
   puts( "Task2 is running and about to delete itself" );
   vTaskDelete( NULL );
}
/*-----------------------------------------------------------*/
```

Перед запуском планировщика создается Задача 1 с приоритетом 1. В теле Задачи 1 динамически создается Задача 2 с более высоким приоритетом. Задача 2 сразу же после создания получает управление, сигнализирует о своем выполнении и сама себя уничтожает. После чего снова управление получает Задача 1.

Следует обратить внимание на тело функции Задачи 2. В нем отсутствует бесконечный цикл, что вполне допустимо, так как функция завершается вызовом API-функции уничтожения этой задачи. Задача 2 в отличие от Задачи 1 является спорадической [5].

Разделение процессорного времени между задачами в учебной программе № 2 показано на рис. 3, а результат ее выполнения — на рис. 4.

Задача 2 существует в системе на протяжении короткого промежутка времени, пока она выполняется. Таким образом, используя

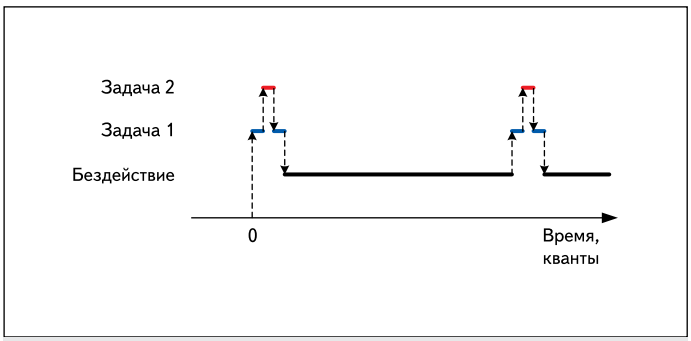

Рис. 3. Разделение процессорного времени между задачами в учебной программе № 2

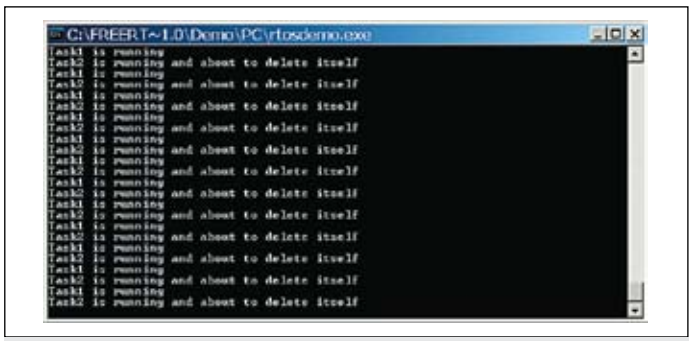

Рис. 4. Результат выполнения учебной программы № 2

динамическое создание/уничтожение задач в реальной программе, удастся достичь экономии памяти, так как память не задействуется под потребности задачи, пока полезные действия этой задачи не нужны.

Выделение памяти при создании задачи

Каждый раз при создании задачи (равно как и при создании других объектов ядра — очередей и семафоров) ядро FreeRTOS выделяет задаче блок памяти из системной кучи — области памяти, доступной для динамического размещения в ней переменных.

Блок памяти, который выделяется задаче, складывается из:

1. Стека задачи. Задается как параметр API-функции ***xTaskCreate()*** при создании задачи.
2. Блока управления задачей (Task Control Block), который представлен структурой tskTCB и содержит служебную информацию, используемую ядром. Размер структуры tskTCB зависит от:
   - настроек FreeRTOS;
   - платформы, на которой она выполняется;
   - используемого компилятора.

Размер блока памяти, который выделяется задаче, на этапе выполнения программы полностью определяется размером отводимого задаче стека, так как размер структуры tskTCB жестко задан на этапе компиляции программы и остается неизменным во время ее выполнения.

Получить точный размер структуры tskTCB для конкретных условий можно, например, добавив в текст программы следующую инструкцию:

```
printf("%d", sizeof(tskTCB));
```

И далее следует прочесть ее размер с какого-либо устройства вывода (в данном случае — с дисплея). При этом нужно учесть, что, так как структура tskTCB используется ядром в собственных целях, то доступа к этой структуре из текста прикладных исходных файлов (main.c в том числе) изначально нет. Чтобы получить доступ к структуре tskTCB, необходимо включить в исходный файл строку:

```
#include "..\\tasks.c"
```

Для учебных программ, приводимых в этой статье и ранее [1], размер структуры tskTCB составляет 70 байт.

Схемы выделения памяти

Функции динамического выделения/освобождения памяти ***malloc()*** и ***free()***, входящие в стандартную библиотеку языка Си, в большинстве случаев не могут напрямую использоваться ядром FreeRTOS, так как их использование сопряжено с рядом проблем:

- Они не всегда доступны в упрощенных компиляторах для микроконтроллеров.
- Их реализация достаточно громоздка, что приводит к дополнительному расходу памяти программ.
- Они редко являются реентерабельными [6], то есть одновременный вызов этих функций из нескольких задач может привести к непредсказуемым результатам.
- Время их выполнения не является детерминированным, то есть от вызова к вызову оно будет меняться, например, в зависимости от степени фрагментации кучи.
- Они могут усложнить конфигурацию компоновщика.

Разные приложения предъявляют различные требования к объему выделяемой памяти и временным задержкам при ее выделении. Поэтому единую схему выделения памяти невозможно применить ко всем платформам, на которые портирована FreeRTOS. Вот почему реализация алгоритма выделения памяти не входит в состав ядра, а выделена в платформенно-зависимый код (в директорию ***\Source\portable\MemMang***). Это позволяет реализовать свой собственный алгоритм выделения памяти для конкретной платформы.

Когда ядро FreeRTOS запрашивает память для своих нужд, происходит вызов API-функции ***pvPortMalloc()***, когда память освобождается — происходит вызов ***vPortFree()***. API-функции ***pvPortMalloc()*** и ***vPortFree()*** имеют такие же прототипы, как и стандартные функции ***malloc()*** и ***free()*** [7]. Реализация API-функций ***pvPortMalloc()*** и ***vPortFree()*** и представляет собой ту или иную схему выделения памяти.

Следует отметить, что API-функции ***pvPortMalloc()*** и ***vPortFree()*** можно беспрепятственно использовать и в прикладных целях, выделяя память для хранения своих переменных.

FreeRTOS поставляется с тремя стандартными схемами выделения памяти, которые содержатся соответственно в исходных файлах ***heap_1.c***, ***heap_2.c***, ***heap_3.c***. В дальнейшем будем именовать стандартные схемы выделения памяти согласно именам файлов с исходным кодом, в которых они определены. Разработчику предоставляется возможность использовать любой алгоритм выделения памяти из поставки FreeRTOS или реализовать свой собственный.

Выбор одной из стандартных схем выделения памяти осуществляется в настройках компилятора (или проекта, если используется среда разработки) добавлением к списку файлов с исходным кодом одного из файлов: ***heap_1.c***, ***heap_2.c*** или ***heap_3.c***.

Схема выделения памяти heap_1.c

Часто программа для микроконтроллера допускает только создание задач, очередей и семафоров и делает это перед запуском планировщика. В этом случае память динамически выделяется перед началом выполнения задач и никогда не освобождается. Такой подход позволяет исключить такие потенциальные проблемы при динамическом выделении памяти, как отсутствие детерминизма и фрагментация, что важно для обеспечения заданного времени реакции системы на внешнее событие.

Схема ***heap_1.c*** предоставляет очень простую реализацию API-функции ***pvPortMalloc()*** и не содержит реализации API-функции ***vPortFree()***. Поэтому такую схему следует использовать, если задачи в программе никогда не уничтожаются. Время выполнения API-функции ***pvPortMalloc()*** в этом случае является детерминированным.

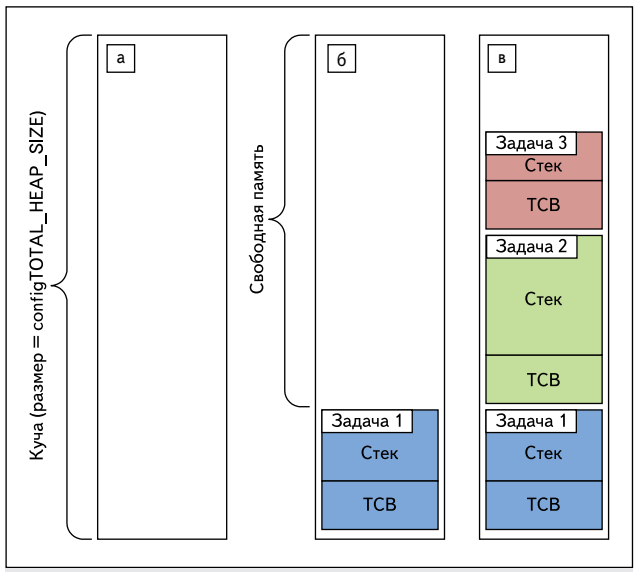

Рис. 5. Распределение памяти кучи при использовании схемы heap_1.c

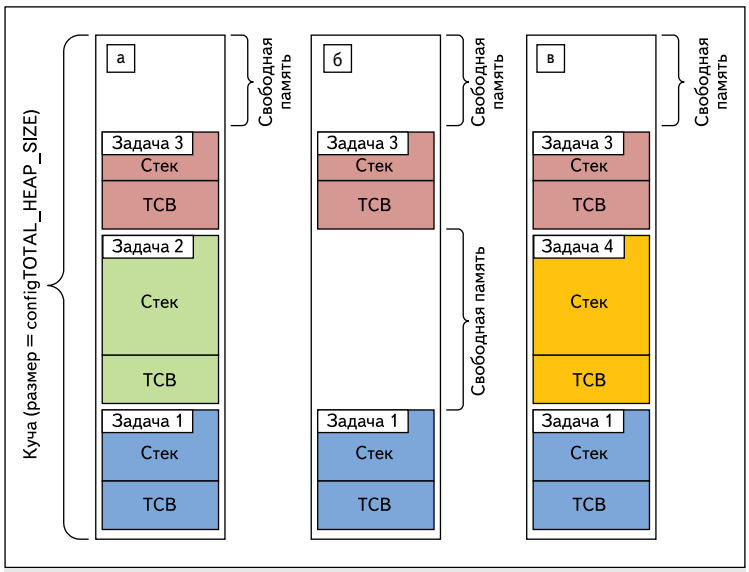

Рис. 6. Распределение памяти кучи при использовании схемы heap_2.c

Вызов ***pvPortMalloc()*** приводит к выделению блока памяти для размещения структуры tskTCB и стека задачи из кучи FreeRTOS. Выделяемые блоки памяти располагаются последовательно друг за другом (рис. 5). Куча FreeRTOS представляет собой массив байт, определенный как обычная глобальная переменная. Размер этого массива в байтах задается макроопределением ***configTOTAL_HEAP_SIZE*** в файле ***FreeRTOSConfig.h***.

Разработчик должен учесть, что объем доступной памяти для размещения переменных, связанных с решением прикладных задач, уменьшается на размер кучи FreeRTOS. Поэтому размер кучи FreeRTOS следует задавать минимальным, но достаточным для размещения всех объектов ядра. Далее будет показано, как получить объем оставшейся свободной памяти в куче FreeRTOS на этапе выполнения программы.

На рис. 5а изображена куча FreeRTOS в момент, когда ни одна задача еще не создана. На рис. 5б и в отображено размещение блоков памяти задач при их последовательном создании и, соответственно, уменьшение объема свободной памяти кучи.

Очевидно, что за счет того, что задачи не уничтожаются, эффект фрагментации памяти кучи исключен.

Схема выделения памяти heap_2.c

Как и в схеме ***heap_1.c***, память для задач выделяется из кучи FreeRTOS размером ***configTOTAL_HEAP_SIZE*** байт. Однако схема ***heap_2.c*** в отличие от ***heap_1.c*** позволяет уничтожать задачи после запуска планировщика, соответственно, она содержит реализацию API-функции ***vPortFree()***.

Так как задачи могут уничтожаться, то блоки памяти, которые они использовали, будут освобождаться, следовательно, в куче может находиться несколько отдельных участков свободной памяти (фрагментация). Для нахождения подходящего участка свободной памяти, в который с помощью API-функции ***pvPortMalloc()*** будет помещен, например, блок памяти задачи, используется алгоритм наилучших подходящих фрагментов (the best fit algorithm).

Работа алгоритма наилучших подходящих фрагментов заключается в следующем. Когда ***pvPortMalloc()*** запрашивает блок памяти заданного размера, происходит поиск свободного участка, размер которого как можно ближе к размеру запрашиваемого блока и, естественно, больше его. Например, структура кучи представляет собой 3 свободных участка памяти размером 5, 25 и 100 байт. Функция ***pvPortMalloc()*** запрашивает блок памяти 20 байт. Тогда наименьший подходящий по размеру участок памяти — участок размером 25 байт. 20 байт из этого участка будут выделены, а оставшиеся 5 байт останутся свободными.

Реализация алгоритма наилучших подходящих фрагментов в FreeRTOS не предусматривает слияния двух примыкающих друг к другу свободных участков в один большой свободный участок. Поэтому при использовании схемы ***heap_2.c*** возможна фрагментация кучи. Однако фрагментации можно не опасаться, если размер выделяемых и освобождаемых впоследствии блоков памяти не изменяется в течение выполнения программы.

Схема выделения памяти ***heap_2.c*** подходит для приложений, где создаются и уничтожаются задачи, причем размер стека при создании задач целесообразно оставлять неизменным.

На рис. 6а изображена куча FreeRTOS, блоки памяти под три задачи располагаются последовательно. На рис. 6б Задача 2 уничтожена, куча содержит два свободных участка памяти. На рис. 6в создана Задача 4 с размером стека таким же, как был у Задачи 2. В соответствии с алгоритмом наилучших подходящих фрагментов Задаче 4 выделен блок, который раньше занимала Задача 2, фрагментации кучи не произошло.

Время выполнения функций ***pvPortMalloc()*** и ***vPortFree()*** для схемы ***heap_2.c*** не является детерминированной величиной, однако их реализация значительно эффективнее стандартных функций ***malloc()*** и ***free()***.

Более подробно с существующими алгоритмами выделения памяти можно познакомиться в [8].

Схема выделения памяти heap_3.c

Схема ***heap_3.c*** использует вызовы функций выделения/освобождения памяти ***malloc()*** и ***free()*** из стандартной библиотеки языка Си. Однако с помощью останова планировщика на время выполнения этих функций достигается псевдореентерабельность (thread safe) этих функций, то есть предотвращается одновременный вызов этих функций из разных задач.

Макроопределение ***configTOTAL_HEAP_SIZE*** не влияет на размер кучи, который теперь задается настройками компоновщика.

Получение объема свободной памяти кучи

Начиная с версии V6.0.0 в FreeRTOS добавлена API-функция ***xPortGetFreeHeapSize()***, с помощью которой можно получить объем доступной для выделения свободной памяти кучи. Ее прототип:

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

Однако следует учесть, что API-функция ***xPortGetFreeHeapSize()*** доступна только при

использовании схем ***heap_1.c*** и ***heap_2.c***. При использовании схемы ***heap_3.c*** получение объема доступной памяти становится нетривиальной задачей.

Резюме по вытесняющей многозадачности в FreeRTOS

Подведя итог по вытесняющей многозадачности в FreeRTOS, можно выделить следующие основные принципы:

1. Каждой задаче назначается приоритет.
2. Каждая задача может находиться в одном из нескольких состояний (выполнение, готовность к выполнению, блокированное состояние, приостановленное состояние).
3. В один момент времени только одна задача может находиться в состоянии выполнения.
4. Планировщик переводит в состояние выполнения готовую к выполнению задачу с наивысшим приоритетом.
5. Задачи могут ожидать наступления события, находясь в блокированном состоянии.
6. События могут быть двух основных типов:
   - временные события;
   - события синхронизации.
7. Временные события чаще всего связаны с организацией периодического выполнения каких-либо полезных действий или с отсчетом времени тайм-аута.
8. События синхронизации чаще всего связаны с обработкой асинхронных событий внешнего мира, например, с получением информации от периферийных (по отношению к процессору) устройств.

Такая схема называется вытесняющим планированием с фиксированными приоритетами (Fixed Priority Preemptive Scheduling). Говорят, что приоритеты фиксированы, потому что планировщик самостоятельно не может изменить приоритет задачи, как это происходит при динамических алгоритмах планирования [5]. Приоритет задаче назначается в явном виде при ее создании, и так же в явном виде он может быть изменен этой же или другой задачей. Таким образом, программист целиком и полностью контролирует приоритеты задач в системе.

Стратегия назначения приоритетов задачам

Как было сказано ранее, программа, выполняющаяся под управлением FreeRTOS, представляет собой совокупность взаимодействующих задач. Чаще всего задача реализуется как какое-либо полезное действие, которое циклически повторяется с заданной частотой/периодом. Каждой задаче назначаются приоритет и частота ее циклического выполнения. Для достижения заданного времени реакции системы на внешние события разработчик должен соответствующим образом назначить приоритеты задачам и определить частоту их выполнения.

Так как FreeRTOS относится к ОСРВ с планированием с фиксированными приоритетами, то рекомендованной стратегией назначения приоритетов является использование принципа «чем меньше период выполнения задачи, тем выше у нее приоритет» (Rate Monotonic Scheduling, RMS) [4, 5].

Основная идея принципа RMS состоит в следующем. Все задачи разделяются по требуемому времени реакции на соответствующее задаче внешнее событие. Каждой задаче назначается уникальный приоритет (то есть в программе не должно быть двух задач с одинаковым приоритетом), причем приоритет тем выше, чем короче время реакции задачи на событие. Частота выполнения задачи устанавливается тем больше, чем больше ее приоритет. Таким образом, самой «ответственной» задаче назначаются наивысший приоритет и наибольшая частота выполнения.

Принцип RMS гарантирует, что система будет иметь детерминированное время реакции на внешнее событие [5]. Однако тот факт, что задачи могут изменять свой приоритет и приоритет других задач во время выполнения, и то, что не все задачи реализуются как циклически выполняющиеся, делают это утверждение в общем случае неверным, и разработчик вынужден прибегать к дополнительным мерам обеспечения заданного времени реакции.

Кооперативная многозадачность во FreeRTOS

До этого момента при изучении FreeRTOS мы использовали режим работы ядра с вытесняющей многозадачностью. Тем не менее, кроме вытесняющей, FreeRTOS поддерживает кооперативную и гибридную (смешанную) многозадачность.

Самое весомое отличие кооперативной многозадачности от вытесняющей — то, что планировщик не получает управление каждый системный квант времени. Вместо этого тело функции, реализующей задачу, должно содержать явный вызов API-функции планировщика ***taskYIELD()***.

Результатом вызова ***taskYIELD()*** может быть как переключение на другую задачу, так и отсутствие переключения, если других задач, готовых к выполнению, нет. Вызов API-функции, которая переводит задачу в блокированное состояние, также приводит к вызову планировщика.

Следует отметить, что отсчет квантов времени ядро FreeRTOS выполняет при использовании любого типа многозадачности, поэтому API-функции, связанные с отсчетом времени, корректно работают и в режиме кооперативной многозадачности. Как и для вытесняющей, в случае применения кооперативной многозадачности каждой задаче необходим собственный стек для хранения своего контекста.

Преимущества кооперативной многозадачности:

1. Меньшее потребление памяти стека при переключении контекста задачи, соответственно, более быстрое переключение контекста. С точки зрения компилятора вызов планировщика «выглядит» как вызов функции, поэтому в стеке автоматически сохраняются регистры процессора и нет необходимости их повторного сохранения в рамках сохранения контекста задачи.
2. Существенно упрощается проблема совместного доступа нескольких задач к одному аппаратному ресурсу. Например, не нужно опасаться, что несколько задач одновременно будут модифицировать одну переменную, так как операция модификации не может быть прервана планировщиком.

Недостатки:

1. Программист должен в явном виде вызывать API-функцию ***taskYIELD()*** в теле задачи, что увеличивает сложность программы.
2. Одна задача, которая по каким-либо причинам не вызвала API-функцию ***taskYIELD()***, приводит к «зависанию» всей программы.
3. Трудно гарантировать заданное время реакции системы на внешнее событие, так как оно зависит от максимального временного промежутка между вызовами ***taskYIELD()***.
4. Вызов ***taskYIELD()*** внутри циклов может замедлить выполнение программы.

Для выбора режима кооперативной многозадачности необходимо задать значение макроопределения ***configUSE_PREEMPTION*** в файле ***FreeRTOSConfig.h*** равным 0:

```
#define configUSE_PREEMPTION		0
```

Значение ***configUSE_PREEMPTION***, равное 1, дает предписание ядру FreeRTOS работать в режиме вытесняющей многозадачности.

Если включить режим кооперативной многозадачности в учебной программе № 1 [1, № 4] так, как показано выше, выполнить сборку проекта и запустить на выполнение полученный исполнимый файл ***rtosdemo.exe***, то можно наблюдать ситуацию, когда все время выполняется один экземпляр задачи, а второй никогда не получает управления (рис. 8, см. КиТ № 4`2011, стр. 98).

Это происходит из-за того, что планировщик никогда не получает управления и не может запустить на выполнение другую задачу. Теперь обязанность запуска планировщика ложится на программиста.

Если добавить в функцию, реализующую задачу, явный вызов планировщика API-функцией ***taskYIELD()***:

```
/*-----------------------------------------------------------*/
/* Функция, реализующая задачу */
void vTask( void *pvParameters )
{
   volatile long ul;
   volatile TaskParam *pxTaskParam;

   /* Преобразование типа void* к типу TaskParam* */
   pxTaskParam = (TaskParam *) pvParameters;

   for( ;; )
   {
      /* Вывести на экран строку, переданную в качестве
      параметра при создании задачи */
      puts( (const char*)pxTaskParam->string );
      /* Задержка на некоторый период T2*/
      for( ul = 0; ul < pxTaskParam->period; ul++ )
      {
      }
      /* Принудительный вызов планировщика.
      Другой экземпляр задачи получит управление
      и будет выполняться, пока не вызовет taskYIELD()
      или блокирующую API-функцию */
      taskYIELD();

   }
   vTaskDelete( NULL );
}
```

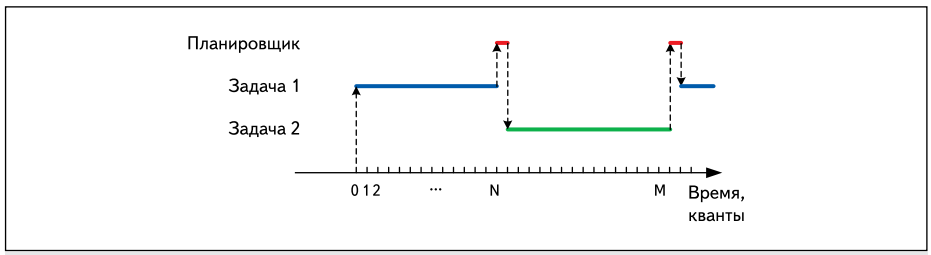

Рис. 7. Разделение процессорного времени между задачами при кооперативной многозадачности при явном вызове taskYIELD()

то процессорное время теперь будут получать оба экземпляра задачи. Результат выполнения программы не будет отличаться от приведенного на рис. 6 (см. КиТ № 4'2011, стр. 98). Но разделение процессорного времени между задачами будет происходить иначе (рис. 7).

На рис. 7 видно, что теперь Задача 1, как только начала выполняться, захватывает процессор на длительное время, до тех пор пока в явном виде не вызовет планировщик API-функцией ***taskYIELD()*** (момент времени N). После вызова планировщика он передает управление Задаче 2, которая тоже удерживает процессор в своем распоряжении до вызова ***taskYIELD()*** (момент времени M). Планировщик теперь не вызывается каждый квант времени, а «ждет», когда его вызовет одна из задач.

Гибридная многозадачность во FreeRTOS

Гибридная многозадачность сочетает в себе автоматический вызов планировщика каждый квант времени, а также возможность принудительного, явного вызова планировщика. Полезной гибридная многозадачность может оказаться, когда необходимо сократить время реакции системы на прерывание. В этом случае в конце тела обработчика прерывания производят вызов планировщика, что приводит к переключению на задачу, ожидающую наступления этого прерывания.

API-функция ***portYIELD_FROM_ISR()*** служит для вызова планировщика из тела обработчика прерывания. Более подробно о ней будет рассказано позже, при изучении двоичных семафоров.

Какого-либо специального действия для включения режима гибридной многозадачности не существует. Достаточно разрешить вызов планировщика каждый квант времени (макроопределение ***configUSE_PREEMPTION*** в файле ***FreeRTOSConfig.h*** должно быть равным 1) и в явном виде вызывать планировщик в функциях, реализующих задачи, и в обработчиках прерываний с помощью API-функций ***taskYIELD()*** и ***portYIELD_FROM_ISR()*** соответственно.

Вытесняющая многозадачность без разделения времени

Ее идея заключается в том, что вызов планировщика происходит только в обработчиках прерываний. Задача выполняется до тех пор, пока не произойдет какое-либо прерывание. После чего она вытесняется задачей, ответственной за обработку внешнего события, связанного с этим прерыванием. Таким образом, задачи не сменяют друг друга по прошествии кванта времени, это происходит только по внешнему событию.

Такой тип многозадачности более эффективен в отношении производительности, чем вытесняющая многозадачность с разделением времени. Процессорное время не тратится впустую на выполнение кода планировщика каждый квант времени.

Для использования этого типа многозадачности макроопределение ***configUSE_PREEMPTION*** в файле ***FreeRTOSConfig.h*** должно быть равным 0 и каждый обработчик прерывания должен содержать явный вызов планировщика ***portYIELD_FROM_ISR()***.

Выводы

На этом изучение задачи как базовой единицы программы, работающей под управлением FreeRTOS, можно считать завершенным. Каждая задача представляет собой отдельную подпрограмму, которая работает независимо от остальных. Однако задачи не могут функционировать изолированно. Они должны обмениваться информацией и координировать свою совместную работу. Во FreeRTOS основным средством обмена информацией между задачами и средством синхронизации задач является механизм очередей.

Поэтому следующие публикации будут посвящены очередям. Подробно будет рассказано:

- как создать очередь;
- каким образом информация хранится и обрабатывается очередью;
- как передать данные в очередь;
- как получить данные из очереди;
- как задачи блокируются, ожидая возможности записать данные в очередь или получить их оттуда;
- какой эффект оказывает приоритет задач при записи и чтении данных в/из очереди.

Литература

1. Курниц А. FreeRTOS — операционная система для микроконтроллеров // Компоненты и технологии. 2011. № 2–4.
2. Barry R. Using the freertos real time kernel: A Practical Guide. 2009.
3. http://www.freertos.org
4. http://en.wikipedia.org/wiki/Rate-monotonic_scheduling
5. http://www.4stud.info/rtos/lecture3.html
6. http://ru.wikipedia.org/wiki/Реентерабельность
7. http://ru.wikipedia.org/wiki/Malloc
8. http://peguser.narod.ru/translations/files/tlsf.pdf