Продолжение. Начало в № 2`2011

FreeRTOS — операционная система для микроконтроллеров

Андрей КУРНИЦ
kurnits@stim.by

В предыдущих статьях [1] читатель познакомился с операционной системой реального времени (ОСРВ) для микроконтроллеров (МК) FreeRTOS. В данной статье будет продолжено изучение базовой единицы любой программы, работающей под управлением FreeRTOS, — задачи. Будет рассказано, как передать в задачу в момент ее создания произвольный параметр и как создать несколько экземпляров одной задачи. Будет показано, как блокировать задачу на определенное время и заставить ее циклически выполняться с заданной частотой. Автор использует удобную для демонстрации возможностей FreeRTOS платформу — порт FreeRTOS для x86 совместимых процессоров.

Рис. 1. Включение 16-разрядного компилятора

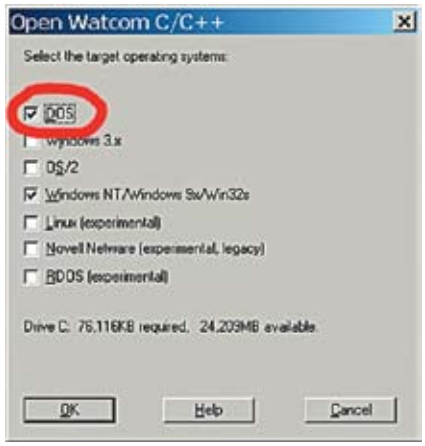

Рис. 2. Включение DOS в список целевых ОС

Подготовка к выполнению FreeRTOS на платформе x86

В предыдущей части [1] был приведен пример создания простой программы, работающей под управлением FreeRTOS. Платформой служил МК фирмы AVR ATmega128. Продолжать подробное рассмотрение и демонстрацию возможностей FreeRTOS на платформе реального МК не всегда удобно. Гораздо удобнее использовать в качестве платформы любой x86 совместимый настольный компьютер, используя соответствующий порт FreeRTOS. Все последующие примеры будут приведены для порта для x86 совместимых процессоров, работающих в реальном режиме. Мы используем бесплатный пакет Open Watcom, включающий Си-компилятор и среду разработки [2], об особенностях установки которого будет сказано ниже. Получаемые в результате компиляции и сборки исполнимые (exe) файлы могут

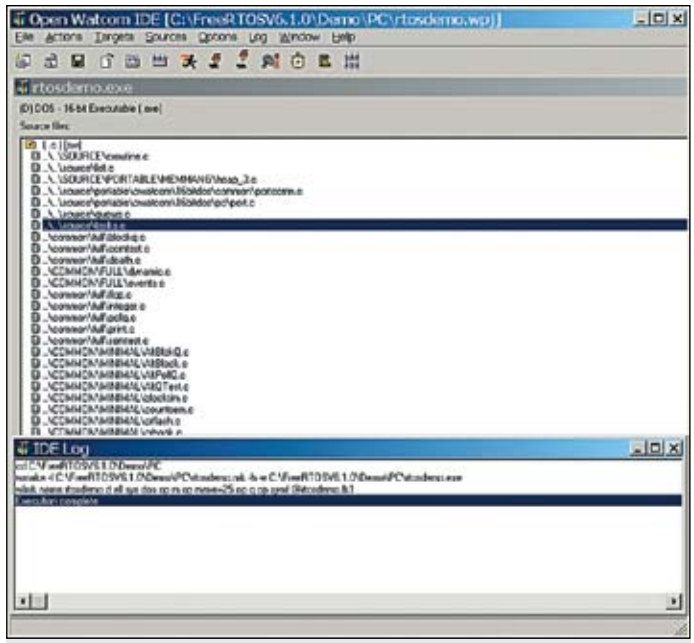

Рис. 3. Успешная сборка демонстрационного проекта в среде Open Watcom

быть выполнены из интерпретатора команд Windows (***cmd.exe***). В качестве альтернативы можно использовать бесплатный эмулятор ОС DOS под названием DOSBox, который позволит выполнять примеры не только из-под Windows, но и из-под UNIX-подобных (FreeBSD, Fedora, Gentoo Linux) и некоторых других ОС [2].

Загрузить последнюю версию пакета Open Watcom можно с официального сайта [2]. На момент написания статьи это версия 1.9. Файл для скачивания: ***open-watcom-c-win32-1.9.exe***. Во время инсталляции пакета следует включить в установку 16-разрядный компилятор для DOS и добавить DOS в список целевых ОС (рис. 1 и 2).

После установки пакета Open Watcom нужно выполнить перезагрузку рабочей станции. Далее можно проверить работу компилятора, открыв демонстрационный проект, входящий в дистрибутив FreeRTOS. Проект располагается в ***C:/FreeRTOSV6.1.0/Demo/PC/*** (в случае установки FreeRTOS на диск ***C:/***). Далее следует открыть файл проекта Open Watcom, который называется ***rtosdemo.wpj***, и выполнить сборку проекта, выбрав пункт меню ***Targets -> Make***. Сборка должна пройти без ошибок (рис. 3).

При этом в директории демонстрационного проекта появится исполнимый файл ***rtosdemo.exe***, запустив который можно наблюдать результаты работы демонстрационного проекта в окне интерпретатора команд Windows (рис. 4).

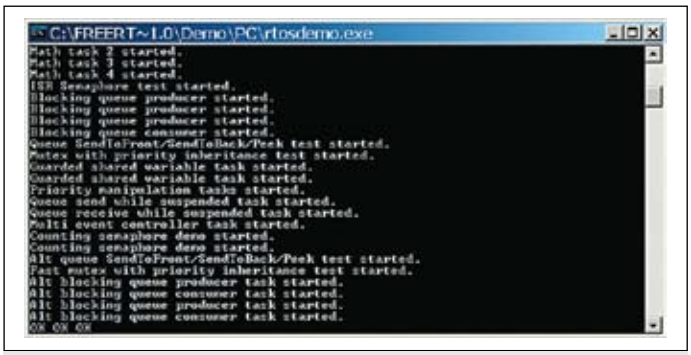

Рис. 4. Работа демонстрационного проекта в среде Windows

В демонстрационный проект включена демонстрация всех возможностей FreeRTOS. Для наших целей, чтобы продолжить изучение задач, не вникая в остальные возможности FreeRTOS, необходимо исключить из проекта все исходные и заголовочные файлы, кроме файлов ядра FreeRTOS и файла ***main.c*** (рис. 5).

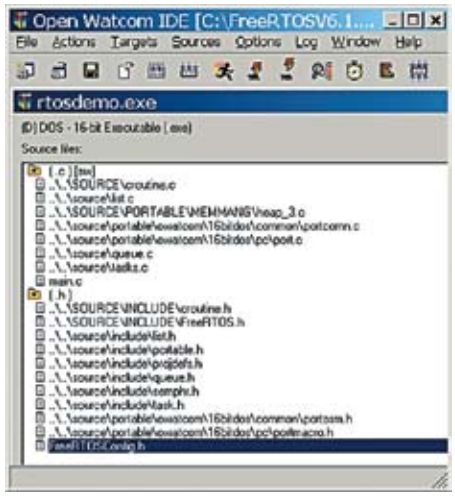

Рис. 5. Минимально необходимый набор исходных и заголовочных файлов в среде Open Watcom

Кроме этого, необходимо произвести настройку ядра, отредактировав заголовочный файл FreeRTOSConfig.h:

```
#ifndef FREERTOS_CONFIG_H
#define FREERTOS_CONFIG_H

#include <i86.h>
#include <conio.h>

#define configUSE_PREEMPTION                    1
#define configUSE_IDLE_HOOK                     0
#define configUSE_TICK_HOOK                     0
#define configTICK_RATE_HZ                      ( ( portTickType ) 1000 )
#define configMINIMAL_STACK_SIZE                ( ( unsigned short ) 256 )
                                                /* This can be made smaller if required. */
#define configTOTAL_HEAP_SIZE                   ( ( size_t ) ( 32 * 1024 ) )
#define configMAX_TASK_NAME_LEN                 ( 16 )
#define configUSE_TRACE_FACILITY                1
#define configUSE_16_BIT_TICKS                  1
#define configIDLE_SHOULD_YIELD                 1
#define configUSE_CO_ROUTINES                   0
#define configUSE_MUTEXES                       1
#define configUSE_COUNTING_SEMAPHORES           1
#define configUSE_ALTERNATIVE_API               1
#define configUSE_RECURSIVE_MUTEXES             1
#define configCHECK_FOR_STACK_OVERFLOW          0 /* Do not use this option on the PC port. */
#define configUSE_APPLICATION_TASK_TAG          1
#define configQUEUE_REGISTRY_SIZE               0

#define configMAX_PRIORITIES                    ( ( unsigned portBASE_TYPE ) 10 )
#define configMAX_CO_ROUTINE_PRIORITIES ( 2 )

/* Set the following definitions to 1 to include the API function, or zero
to exclude the API function. */

#define INCLUDE_vTaskPrioritySet                1
#define INCLUDE_uxTaskPriorityGet               1
#define INCLUDE_vTaskDelete                     1
#define INCLUDE_vTaskCleanUpResources           1
#define INCLUDE_vTaskSuspend                    1
#define INCLUDE_vTaskDelayUntil                 1
#define INCLUDE_vTaskDelay                      1
#define INCLUDE_uxTaskGetStackHighWaterMark 0 /* Do not use this option on the PC port. */

#endif /* FREERTOS_CONFIG_H */
```

Передача параметра в задачу при ее создании

На этом подготовительный этап можно считать завершенным. Как говорилось в [1], при создании задачи с помощью API-функции ***xTaskCreate()*** есть возможность передать в функцию, реализующую задачу, произвольный параметр.

Разработаем учебную программу № 1, которая будет создавать два экземпляра одной задачи. Чтобы каждый экземпляр задачи выполнял уникальное действие, передадим в качестве параметра строку символов и значение периода, которое будет сигнализировать о том, что задача выполнена. Для этого следует отредактировать файл ***main.c***:

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

/* Структура, содержащая передаваемую в задачу информацию */
typedef struct TaskParam_t {
   char           string[32];   /* строка */
   long           period;       /* период */
} TaskParam;


/* Объявление двух структур TaskParam */
TaskParam xTP1, xTP2;


/*-----------------------------------------------------------*/
/* 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++ )
        {
        }

    }
    vTaskDelete( NULL );
}

/*-----------------------------------------------------------*/
/* Точка входа. С функции main() начнется выполнение программы. */
short main( void )
{
    /* Заполнение полей структуры, передаваемой Задаче 1 */
    strcpy(xTP1.string, "Task 1 is running");
    xTP1.period = 10000000L;

    /* Заполнение полей структуры, передаваемой Задаче 2 */
    strcpy(xTP2.string, "Task 2 is running");
    xTP2.period = 30000000L;

    /* Создание Задачи 1. Передача ей в качестве параметра указателя на структуру xTP1 */
    xTaskCreate(  vTask,   /* Функция, реализующая задачу */
                           ( signed char * ) "Task1",
                           configMINIMAL_STACK_SIZE,
                           (void*)&xTP1, /* Передача параметра */
                           1,
                           NULL );

    /* Создание Задачи 2. Передача ей указателя на структуру xTP2 */
    xTaskCreate( vTask, ( signed char * ) "Task2", configMINIMAL_STACK_SIZE, (void*)&xTP2, 1, NULL );

    /* Запуск планировщика */
    vTaskStartScheduler();

    return 1;
}
```

Выполнив сборку проекта и запустив на выполнение полученный исполнимый файл ***rtosdemo.exe***, можно наблюдать результат работы учебной программы № 1 (рис. 6).

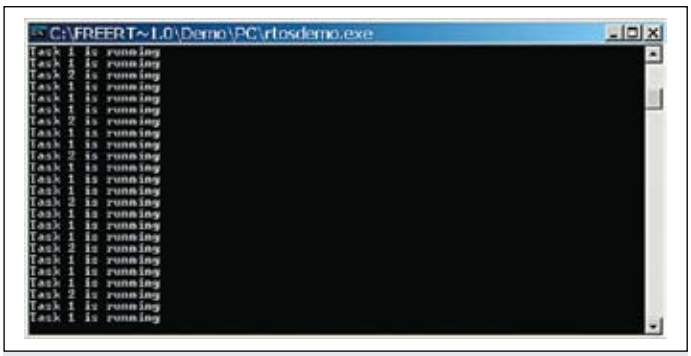

Рис. 6. Результат выполнения учебной программы № 1 в среде Windows

Задача 2 выводит сообщение о своей работе в три раза реже, чем Задача 1. Это объясняется тем, что в Задачу 2 было передано значение периода в 3 раза большее, чем в Задачу 1. Таким образом, передача различных параметров в задачи при их создании позволила добиться различной функциональности отдельных экземпляров одной задачи.

Приоритеты задач

В предыдущей статье [1] читатель познакомился с механизмом приоритетов задач. Далее будет показано, как значение приоритета влияет на выполнение задачи.

При создании задачи ей назначается приоритет. Приоритет задается с помощью параметра ***uxPriority*** функции ***xTaskCreate()***. Максимальное количество возможных приоритетов определяется макроопределением ***configMAX_PRIORITIES*** в заголовочном файле ***FreeRTOSConfig.h***. В целях экономии ОЗУ необходимо задавать наименьшее, но достаточное значение ***configMAX_PRIORITIES***. Нулевое значение приоритета соответствует наиболее низкому приоритету, значение (***configMAX_PRIORITIES-1***) — наиболее высокому (в ОС семейства Windows наоборот — приоритет 0 наивысший).

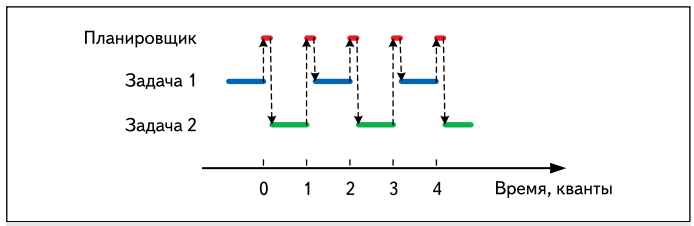

Рис. 7. Разделение процессорного времени между задачами в учебной программе № 1

Планировщик гарантирует, что среди всех задач, находящихся в состоянии готовности к выполнению, перейдет в состояние выполнения та задача, которая имеет наивысший приоритет. Если в программе созданы несколько задач с одинаковым приоритетом, то они будут выполняться в режиме разделения времени [1]. То есть задача выполняется в течение системного кванта времени, после чего планировщик переводит ее в состояние готовности и запускает следующую задачу с таким же приоритетом, и далее по кругу. Таким образом, задача выполняется за один квант времени и находится в состоянии готовности к выполнению (но не выполняется) в течение стольких квантов времени, сколько имеется готовых к выполнению задач с таким же приоритетом.

На рис. 7 показано, как задачи разделяют процессорное время в учебной программе № 1. Кроме хода выполнения двух задач, на рис. 7 показано выполнение кода планировщика каждый системный квант времени. Выполнение кода планировщика приводит к переключению на следующую задачу с одинаковым приоритетом.

Модифицируем учебную программу № 1 так, чтобы задачам назначался разный приоритет. Пусть Задача 2 получит приоритет, равный 2, а приоритет Задачи 1 останется прежним — равным 1. Для этого следует отредактировать вызов API-функции ***xTaskCreate()*** для создания Задачи 2:

```
…
xTaskCreate( vTask, ( signed char * ) "Task2", configMINIMAL_STACK_SIZE, (void*)&xTP2, 2, NULL );
…
```

Выполнив сборку модифицированной учебной программы и запустив ее на выполнение, можно наблюдать ситуацию, когда все время будет выполняться Задача 2, а Задача 1 никогда не получит управление (рис. 8).

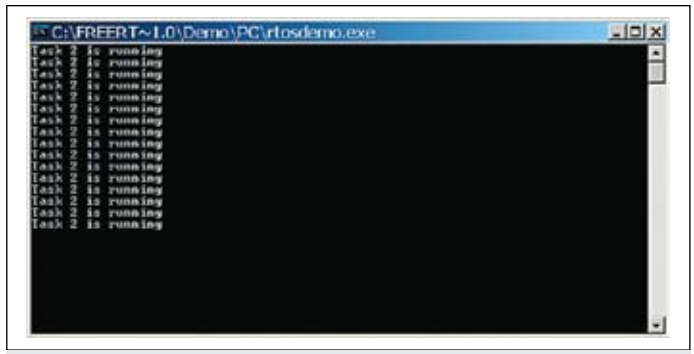

Рис. 8. Результат работы учебной программы в случае назначения Задаче 2 более высокого приоритета

Задача 2, как и Задача 1, все время находится в состоянии готовности к выполнению. За счет того, что Задача 2 имеет приоритет выше, чем Задача 1, каждый квант времени планировщик будет отдавать управление именно ей, а Задача 1 никогда не получит процессорного времени (рис. 9).

Этот пример показывает необходимость пользоваться приоритетами осмотрительно, так как никакого алгоритма старения в планировщике не предусмотрено (как в ОС общего назначения). Поэтому

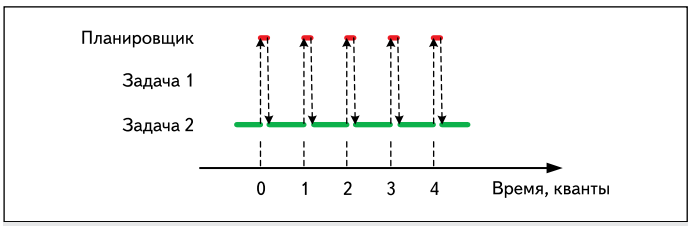

Рис. 9. Разделение времени между задачами, когда Задача 2 имеет более высокий приоритет, чем Задача 1

возможна ситуация «зависания» задачи с низким приоритетом, которая никогда не выполнится. Программисту необходимо тщательно проектировать прикладные программы и благоразумно задавать уровни приоритетов, чтобы избежать такой ситуации. Далее будет показано, как избежать «зависания» низкоприоритетных задач, используя механизм событий для управления ходом их выполнения.

Следует отметить, что FreeRTOS позволяет динамически менять приоритет задачи во время выполнения программы. Для получения и задания приоритета задачи во время выполнения служат API-функции ***uxTaskPriorityGet()*** и ***vTaskPrioritySet()*** соответственно.

Подсистема времени FreeRTOS

Подробнее остановимся на системном кванте времени. Планировщик получает управление каждый квант времени, это происходит по прерыванию от таймера. Продолжительность системного кванта определяется периодом возникновения прерываний от таймера и задается в файле ***FreeRTOSConfig.h*** макроопределением ***configTICK_RATE_HZ***.

configTICK_RATE_HZ определяет частоту отсчета системных квантов в герцах, например значение ***configTICK_RATE_HZ***, равное 100 (Гц), определяет продолжительность системного кванта, равную 10 мс. Следует отметить, что в большинстве демонстрационных проектов продолжительность системного кванта устанавливается равной 1 мс (***configTICK_RATE_HZ*** = 1000).

Все API-функции, связанные с измерением временных интервалов, в качестве единицы измерения времени используют системный квант. Используя макроопределение ***portTICK_RATE_MS***, можно получить продолжительность системного кванта в миллисекундах. Но для задания длительности кванта нужно использовать макроопределение ***configTICK_RATE_HZ***.

Следует также упомянуть о счетчике квантов — это системная переменная типа portTickType, которая увеличивается на единицу по прошествии одного кванта времени и используется ядром FreeRTOS для измерения временных интервалов. Значение счетчика квантов начинает увеличиваться после запуска планировщика, то есть после выполнения функции ***vTaskStartScheduler()***. Текущее значение счетчика квантов может быть получено с помощью API-функции ***xTaskGetTickCount()***.

События как способ управления выполнением задач

В учебных программах, приведенных выше, задачи были реализованы так, что они постоянно нуждались в процессорном времени. Даже когда задача ничего не выводила на экран, она занималась отсчетом времени с помощью пустого цикла ***for***.

Такая реализация задачи целесообразна только при назначении задаче самого низкого приоритета. В противном случае наличие такой постоянно готовой к выполнению задачи с довольно высоким приоритетом приведет к тому, что другие задачи, имеющие более низкий приоритет, никогда не будут выполняться.

Гораздо эффективнее управлять выполнением задач с помощью событий. Управляемая событием задача выполняется только после того, как некоторое событие произошло. Если событие не произошло и задача ожидает его наступления, то она НЕ находится в состоянии ГОТОВНОСТИ к выполнению, а следовательно, не может быть выполнена планировщиком. Планировщик распределяет процессорное время только между задачами, ГОТОВЫМИ к выполнению. Таким образом, если высокоприоритетная задача ожидает наступления некоторого события, то есть не находится в состоянии готовности к выполнению, то планировщик отдаст управление готовой к выполнению более низкоприоритетной задаче.

Таким образом, применение событий для управления ходом выполнения задач позволяет создавать программы с множеством различных приоритетов задач, и программист может не опасаться того, что высокоприоритетная задача «заберет» себе все процессорное время.

Блокированное состояние задачи

Если задача ожидает наступления события, то она находится в блокированном состоянии (рис. 5 в КиТ № 3`2011, стр. 111). Во FreeRTOS существуют два вида событий:

1. Временное событие — это событие, связанное с истечением временного промежутка или наступлением определенного момента абсолютного времени. Например, задача может войти в блокированное состояние, пока не пройдет 10 мс.
2. Событие синхронизации (внешнее по отношению к задаче) — это событие, которое генерируется в другой задаче или в теле обработчика прерывания МК. Например, задача блокирована, когда ожидает появления данных в очереди. Данные в очередь поступают от другой задачи.

События синхронизации могут быть связаны с множеством объектов ядра, такими как очереди, двоичные и счетные семафоры, рекурсивные семафоры и мьютексы, которые будут описаны в дальнейших публикациях.

Во FreeRTOS есть возможность заблокировать задачу, заставив ее ожидать события синхронизации, но определить при этом тайм-аут ожидания. То есть выход задачи из блокированного состояния возможен как при наступлении события синхронизации, так и по прошествии времени тайм-аута, если событие синхронизации так и не произошло. Например, задача ожидает появления данных из очереди. Тайм-аут при этом установлен равным 10 мс. В этом случае выход задачи из блокированного состояния возможен при выполнении двух условий:

- Данные в очередь поступили.
- Данные не поступили, но вышло время тайм-аута, равное 10 мс.

Реализация задержек с помощью API-функции vTaskDelay()

Вернемся к рассмотрению учебной программы № 1. Задачи в этой программе выполняли полезное действие (в нашем случае — вывод текстовой строки на экран), после чего ожидали определенный промежуток времени, то есть выполняли задержку на какое-то время. Реализация задержки в виде пустого цикла не эффективна. Один из основных недостатков мы продемонстрировали, когда задачам был назначен разный приоритет. А именно, когда высокоприоритетная задача все время остается в состоянии готовности к выполнению (не переходит ни в блокированное, ни в приостановленное состояние), она поглощает все процессорное время, и низкоприоритетные задачи никогда не выполняются.

Для корректной реализации задержек средствами FreeRTOS следует применять API-функцию ***vTaskDelay()***, которая переводит задачу, вызывающую эту функцию, в блокированное состояние на заданное количество квантов времени. Ее прототип:

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

Единственным аргументом является ***xTicksToDelay***, который непосредственно задает количество квантов времени задержки.

Например, пусть задача вызвала функцию ***xTicksToDelay(100)*** в момент времени, когда счетчик квантов был равен 5000. Задача сразу же блокируется, планировщик отдаст управление другой задаче, а вызывающая задача вернется в состояние готовности к выполнению, только когда счетчик квантов достигнет значения 5100. В течение времени, пока счетчик квантов будет увеличиваться от 5000 до 5100, планировщик будет выполнять другие задачи, в том числе задачи с более низким приоритетом.

Следует отметить, что программисту нет необходимости отслеживать переполнение счетчика квантов времени. API-функции, связанные с отсчетом времени (в том числе и ***vTaskDelay()***), берут эту обязанность на себя.

Рассмотрим учебную программу № 2, которая выполняет те же функции, что и программа № 1, но для создания задержек в ней используется API-функция ***vTaskDelay()***. Кроме того, задаче при ее создании передается не абстрактное значение периода, а значение периода в миллисекундах:

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

/* Структура, содержащая передаваемую в задачу информацию */
typedef struct TaskParam_t {
   char string[32]; /* строка */
   long period; /* период, миллисекунды*/
} TaskParam;

/* Объявление двух структур TaskParam */
TaskParam xTP1, xTP2;

/*-----------------------------------------------------------*/
/* Функция, реализующая задачу */
void vTask( void *pvParameters )
{
   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 время, заданное в миллисекундах */
      /* pxTaskParam->period задан в миллисекундах /*
      /* Разделив его на кол-во мс в кванте, получим кол-во квантов */
      vTaskDelay(pxTaskParam->period / portTICK_RATE_MS);
   }
   vTaskDelete( NULL );
}

/*-----------------------------------------------------------*/
/* Точка входа. С функции main() начнется выполнение программы. */
short main( void )
{
   /* Заполнение полей структуры, передаваемой Задаче 1 */
   strcpy(xTP1.string, "Task 1 is running");
   xTP1.period = 1000L;       /* 1000 мс */

   /* Заполнение полей структуры, передаваемой Задаче 2 */
   strcpy(xTP2.string, "Task 2 is running");
   xTP2.period = 3000L;       /* 3000 мс */

   /* Создание Задачи 1 с приоритетом 1. Передача ей в качестве
параметра указателя на структуру xTP1 */
   xTaskCreate( vTask, ( signed char * ) "Task1", configMINIMAL_
STACK_SIZE, (void*)&xTP1, 1, NULL );

   /* Создание Задачи 2 с приоритетом 2. Передача ей указателя
на структуру xTP2 */
   xTaskCreate( vTask, ( signed char * ) "Task2", configMINIMAL_
STACK_SIZE, (void*)&xTP2, 2, NULL );

   /* Запуск планировщика */
   vTaskStartScheduler();

   return 1;
}
```

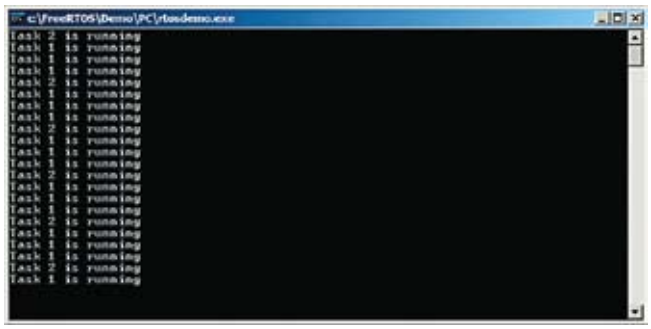

Рис. 10. Результат работы учебной программы № 2

По результатам работы учебной программы № 2 (рис. 10) видно, что процессорное время теперь получает как высокоприоритетная задача 2, так и низкоприоритетная задача 1.

Выполнение задач в учебной программе № 2 приведено на рис. 11. Выполнение кода планировщика в целях упрощения рисунка не приводится. Бóльшую часть времени процессор бездействует, а следовательно, теперь задачи очень экономно расходуют процессорное время.

В момент времени (1) происходит запуск планировщика. На этот момент Задача 1 и Задача 2 находятся в состоянии готовности к выполнению, однако приоритет Задачи 2 выше, поэтому именно ей планировщик передает управление. Задача 2 выполняет полезную работу (выводит строку ***"Task 2 is running"***) (рис. 10), после чего выполняет API-функцию ***vTaskDelay()***, в результате чего Задача 2 переходит в блокированное состояние. После вызова функции ***vTaskDelay()*** выполняемая в данный момент Задача 2 перешла в блокированное состояние и не нуждается в процессорном времени, поэтому для того чтобы занять процессор другой задачей, функция ***vTaskDelay()*** вызывает планировщик. Теперь в списке готовых к выполнению задач осталась только Задача 1, которой планировщик и отдает управление (момент времени (2)). Задача 1 выполняет свою полезную работу: также вызывает API-функцию ***vTaskDelay()*** и переходит в блокированное состояние (момент времени (3)). В этот момент нет ни одной задачи, готовой к выполнению, поэтому планировщик вызывает системную задачу, которая не выполняет никакой полезной работы, — задачу Бездействие (4). Подробнее о задаче Бездействие будет сказано ниже.

На протяжении времени, когда Задача 1 и Задача 2 находятся в блокированном состоянии, кроме выполнения задачи Бездействие, ядро FreeRTOS отсчитывает кванты времени, прошедшие с моментов вызовов API-функции ***vTaskDelay()***. Как только ядро отсчитает 1000 квантов (1000 мс), оно переведет Задачу 1 из блокированного в состояние готовности к выполнению (момент времени (5)). Планировщик отдаст ей управление, она выполнит полезную работу и снова перейдет в блокированное состояние на время 1000 мс и т. д. Задача 2 будет находиться в блокированном состоянии на протяжении 3000 мс. В момент времени (7) из блокированного состояния в состояние готовности к выполнению перейдут обе задачи, однако планировщик запустит (переведет в состояние выполнения) Задачу 2, так как приоритет у нее выше.

API-функция vTaskDelayUntil ()

API-функция ***vTaskDelayUntil()*** служит для тех же целей, что и ***vTaskDelay()***, — для перевода задачи в блокированное состояние на заданное время. Однако она имеет некоторые особенности, позволяющие с меньшими усилиями реализовать циклическое выполнение кода задачи с точно заданным периодом.

Часто перед программистом стоит задача циклического выполнения какого-либо

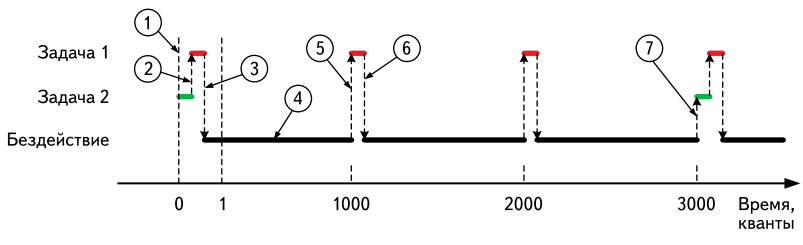

Рис. 11. Разделение процессорного времени между задачами в учебной программе № 2

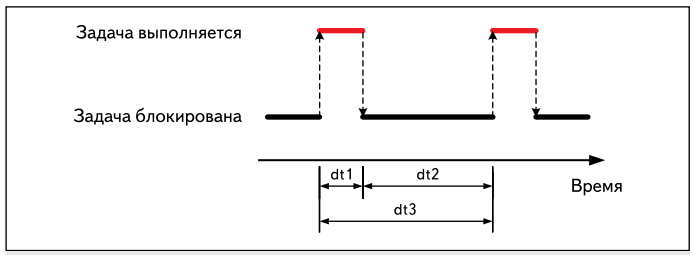

Рис. 12. Ход выполнения циклической задачи. Задержка реализована API-функцией vTaskDelay()

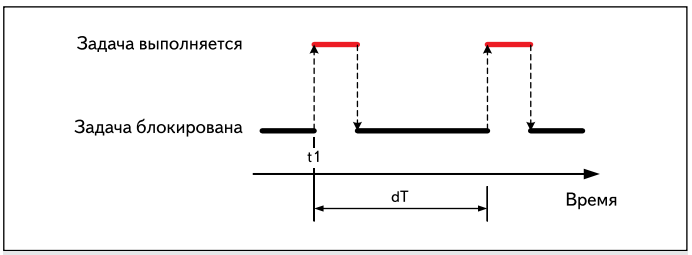

Рис. 13. Ход выполнения циклической задачи. Задержка реализована API-функцией vTaskDelayUntil()

действия с четко фиксированной частотой и, следовательно, периодом. API-функция ***vTaskDelay()*** переводит задачу в блокированное состояние на промежуток времени, который отсчитывается от момента вызова ***vTaskDelay()***. В случае реализации циклически повторяющегося действия период его выполнения *dt*3 будет складываться из времени его выполнения *dt*1 и задержки *dt*2, создаваемой функцией ***vTaskDelay()*** (рис. 12).

Если стоит цель обеспечить циклическое выполнение с точно заданным периодом *dt*3, то необходимо знать время выполнения тела задачи *dt*1, чтобы скорректировать величину задержки *dt*2. Это создает дополнительные сложности.

Для таких целей предназначена API-функция ***vTaskDelayUntil()***. Программист в качестве ее параметра задает период *dT*, который отсчитывается с момента *t*1 — момента выхода задачи из блокированного состояния (рис. 13).

Прототип функции ***vTaskDelayUntil()***:

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

Функции ***vTaskDelayUntil()*** передаются следующие аргументы:

1. ***pxPreviousWakeTime*** — указатель на переменную, в которой хранится значение счетчика квантов в момент последнего выхода задачи из блокированного состояния (момент времени *t*1 на рис. 13). Этот момент используется как отправная точка для отсчета времени, на которое задача переходит в блокированное состояние. Переменная, на которую ссылается указатель ***pxPreviousWakeTime***, автоматически обновляется функцией ***vTaskDelayUntil()***, поэтому при типичном использовании эта переменная не должна модифицироваться в теле задачи. Исключение составляет начальная инициализация, как показано в примере ниже.
2. ***xTimeIncrement*** — непосредственно задает период выполнения задачи. Задается в квантах; для задания в миллисекундах может использоваться макроопределение ***portTICK_RATE_MS***.

Типичное применение API-функции ***vTaskDelayUntil()*** в теле функции, реализующей задачу:

```
/* Функция задачи, которая будет циклически выполняться с жестко заданным периодом в 50 мс */
void vTaskFunction( void *pvParameters )
{
   /* Переменная, которая будет хранить значение счетчика квантов
   в момент выхода задачи из блокированного состояния */
   portTickType xLastWakeTime;
   /* Переменная xLastWakeTime нуждается в инициализации текущим значением счетчика квантов.
   Это единственный случай, когда ее значение задается явно.
   В дальнейшем ее значение будет автоматически модифицироваться API-функцией vTaskDelayUntil(). */
   xLastWakeTime = xTaskGetTickCount();
   /* Бесконечный цикл */
   for( ;; )
   {
       /* Какая-либо полезная работа */
       /* ... */

       /* Период выполнения этой задачи составит 50 мс.
       Разделив это значение на кол-во миллисекунд в 1 кванте portTICK_RATE_MS,
       получим кол-во квантов периода, что и является аргументом vTaskDelayUntil().
       Переменная xLastWakeTime автоматически модифицируется внутри vTaskDelayUntil(),
       поэтому нет необходимости делать это в явном виде. */
       vTaskDelayUntil( &xLastWakeTime, ( 50 / portTICK_RATE_MS ) );
   }
}
```

Задача Бездействие

Пока на МК подано питание и он находится не в режиме энергосбережения, МК должен выполнять какой-либо код. Поэтому хотя бы одна задача должна постоянно находиться в состоянии готовности к выполнению. Однако, как показано в учебной программе № 2, может сложиться ситуация, когда все задачи в программе могут быть заблокированы.

В этом случае МК будет выполнять задачу Бездействие (Idle task). Задача Бездействие создается автоматически при запуске планировщика API-функцией ***vTaskStartScheduler()***. Задача Бездействие постоянно находится в состоянии готовности к выполнению. Ее приоритет задается макроопределением ***tskIDLE_PRIORITY*** как самый низкий в программе (обычно 0). Это гарантирует, что задача Бездействие не будет выполняться, пока в программе есть хотя бы одна задача в состоянии готовности к выполнению. Как только появится любая готовая к выполнению задача, задача Бездействие будет вытеснена ею.

Программисту предоставляется возможность добавить свою функциональность в задачу Бездействие. Для этих целей есть возможность определить функцию-ловушку (***Idle hook function***, которая является функцией обратного вызова — ***Callback function***), реализующую функциональность задачи Бездействие (далее будем называть ее функцией задачи Бездействие). Функция задачи Бездействие отличается от функции, реализующей обычную задачу. Функция задачи Бездействие не содержит бесконечного цикла, а автоматически вызывается ядром FreeRTOS множество раз, пока выполняется задача Бездействие, то есть ее тело помещается внутрь бесконечного цикла средствами ядра.

Добавление своей функциональности в функцию задачи Бездействие окажется полезным в следующих случаях:

1. Для реализации низкоприоритетных фоновых задач.
2. Для измерения резерва МК по производительности. Во время выполнения задачи Бездействие процессор не занят полезной работой, то есть простаивает. Отношение времени простоя процессора ко всему времени выполнения программы даст представление о резерве процессора по производительности, то есть о возможности добавить дополнительные задачи в программу.
3. Для снижения энергопотребления микроконтроллерного устройства. Во многих МК есть возможность перехода в режим пониженного энергопотребления для экономии электроэнергии. Это актуально, например, в случае проектирования устройства с батарейным питанием. Выход из режима энергосбережения во многих МК возможен по прерыванию от таймера. Если настроить МК так, чтобы вход в режим пониженного энергопотребления происходил в теле функции задачи Бездействие, а выход — по прерыванию от того же таймера, что используется ядром FreeRTOS для формирования квантов времени, то это позволит значительно понизить энергопотребление устройства во время простоя процессора.

Есть некоторые ограничения на реализацию функции задачи Бездействие:

1. Задачу Бездействие нельзя пытаться перевести в блокированное или приостановленное состояние.

2. Если программа допускает использование API-функции уничтожения задачи ***vTaskDelete()***, то функция задачи Бездействие должна завершать свое выполнение в течение разумного периода времени. Это требование объясняется тем, что функция задачи Бездействие ответственна за освобождение ресурсов ядра после уничтожения задачи. Таким образом, временная задержка в теле функции задачи Бездействие приведет к такой же задержке в очистке ресурсов, связанных с уничтоженной задачей, и ресурсы ядра не будут освобождены вовремя.

Чтобы задать свою функцию задачи Бездействие, необходимо в файле настройки ядра ***FreeRTOSConfig.h*** задать макроопределение ***configUSE_IDLE_HOOK*** равным 1. В одном из файлов исходного кода должна быть определена функция задачи Бездействие, которая имеет следующий прототип:

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

Значение ***configUSE_IDLE_HOOK***, равное 0, используется, когда не нужно добавлять дополнительную функциональность.

Создадим учебную программу № 3, демонстрирующую использование функции задачи Бездействие. В программе будет определена глобальная переменная-счетчик, задача Бездействие будет инкрементировать значение этой переменной. Также будет создана задача вывода значения переменной-счетчика на экран каждые 250 мс.

Текст учебной программы № 3:

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

/* Глобальная переменная-счетчик, которая будет увеличиваться на 1
 при каждом вызове функции задачи Бездействие */
volatile unsigned long ulIdleCycleCount = 0;

/*-----------------------------------------------------------*/
/* Функция, реализующая задачу вывода на экран значения
ulIdleCycleCount
каждые 250 мс */
void vTaskFunction( void *pvParameters )
{
    /* Бесконечный цикл */
    for( ;; )
    {
        /* Вывести на экран значение переменной ulIdleCycleCount */
        printf( "ulIdleCycleCount = %lu\n\r", ulIdleCycleCount );
        /* Задержка на 250 мс */
        vTaskDelay( 250 / portTICK_RATE_MS );
    }
    vTaskDelete( NULL );
}

/*-----------------------------------------------------------*/
/* Точка входа. С функции main() начнется выполнение программы. */
short main( void )
{
    /* Создание задачи с приоритетом 1. Параметр не передается */
    xTaskCreate( vTaskFunction, ( signed char * ) "Task",
configMINIMAL_STACK_SIZE, NULL, 1, NULL );
    /* Запуск планировщика */
    vTaskStartScheduler();

    return 1;
}

/* Функция, реализующая задачу Бездействие.
 Ее имя ОБЯЗАТЕЛЬНО должно быть vApplicationIdleHook.
 Аргументов не получает. Ничего не возвращает */
void vApplicationIdleHook( void )
{
    /* Увеличить переменную-счетчик на 1 */
    ulIdleCycleCount++;
}
```

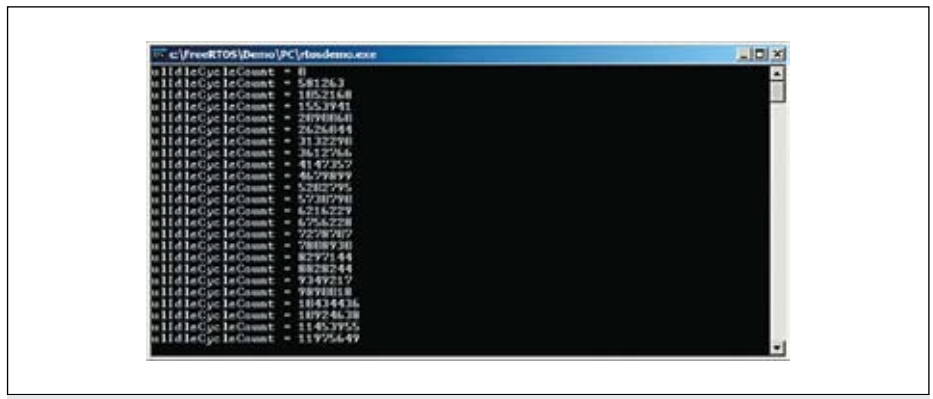

Рис. 14. Результат работы учебной программы № 3

Результат выполнения учебной программы № 3 приведен на рис. 14. Видно, что за 250 мс, пока задача вывода значения на экран пребывает в блокированном состоянии, функция задачи Бездействие «успевает выполниться» большое количество раз.

Учебная программа № 3 затрагивает еще один очень важный аспект написания программ, работающих под управлением ОСРВ, — одновременное использование одного аппаратного ресурса различными задачами. В нашем случае в качестве такого ресурса выступает глобальная переменная, доступ к которой осуществляет как задача Бездействие, так и задача отображения значения этой переменной. При совместном доступе нескольких задач к общей переменной возможна ситуация, когда выполнение одной задачи прерывается планировщиком именно в тот момент, когда задача модифицирует общую переменную, когда та еще содержит не окончательное (искаженное) значение. При этом результат работы другой задачи, которая получит управление и обратится к этой переменной, также окажется искаженным.

Однако в учебной программе № 3 задача Бездействие не может прервать операцию с общей переменной в теле задачи отображения, так как задача Бездействие будет выполняться лишь тогда, когда задача отображения завершит действия с общей переменной (вывод ее на экран функцией ***printf()***) и перейдет в блокированное состояние вызовом API-функции ***vTaskDelay()***. Одновременный совместный доступ, таким образом, исключен. Поэтому дополнительных мер для обеспечения совместного доступа к общему ресурсу в учебной программе № 3 не предпринимается.

Выводы

В статье описан способ передачи произвольного параметра в задачу при ее создании. Внимание было уделено механизму приоритетов и тому, как значение приоритета влияет на ход выполнения задачи. Рассказано о возможностях FreeRTOS для реализации задержек и периодического выполнения задачи. Изучена задача Бездействие и возможности, которые она предоставляет.

В следующих публикациях будет подробно описан процесс принудительного изменения приоритета задач в ходе их выполнения, показано, как динамически создавать и уничтожать задачи. Будет подведен итог по вытесняющей многозадачности во FreeRTOS и рассказано о возможностях кооперативной многозадачности. Далее внимание будет сфокусировано на взаимодействии и передаче информации между задачами и между прерываниями и задачами средствами FreeRTOS. ■

Литература

1. Курниц А. FreeRTOS — операционная система для микроконтроллеров // Компоненты и технологии. 2011. № 2–3.
2. http://www.openwatcom.org/index.php/Download
3. http://www.dosbox.com
4. Barry R. Using the FreeRTOS real time kernel: A Practical Guide. 2009.
5. http://www.freertos.org