Продолжение. Начало в № 2`2011

FreeRTOS — операционная система для микроконтроллеров

Андрей КУРНИЦ
kurnits@stim.by

Эта статья продолжает знакомить читателя с созданием программ, работающих под управлением FreeRTOS — операционной системы для микроконтроллеров. На этот раз речь пойдет о проблемах организации совместного доступа нескольких задач и/или прерываний к одному ресурсу в среде FreeRTOS.

Введение

Статья поможет читателям ответить на следующие вопросы:
1) Что означает термин «ресурс»?
2) Когда и почему необходимо управление доступом к ресурсам?
3) Что такое механизм взаимного исключения и способы его реализации?
4) Что такое критическая секция и способы ее реализации во FreeRTOS?
5) Как применять мьютексы для реализации механизма взаимного исключения?
6) Что такое инверсия приоритетов и как наследование приоритетов позволяет уменьшить (но не устранить) ее воздействие?
7) Другие потенциальные проблемы, возникающие при использовании мьютексов.
8) Задачи-сторожа — создание и использование.
9) Функция, вызываемая каждый системный квант времени.

Ресурсы и доступ к ним

Под ресурсами микроконтроллерной системы понимают как физически существующие устройства внутри микроконтроллера (области оперативной памяти и периферийные устройства), так и внешние по отношению к микроконтроллеру устройства (другие микроконтроллеры, контроллеры протоколов, дисплеи и т. д.). К этим группам можно свести все примеры ресурсов, приводимые ниже.

Потенциальная причина сбоев и ошибок в мультизадачных системах — это неправильно организованный совместный доступ к ресурсам из нескольких задач и/или прерываний. Одна задача получает доступ к ресурсу, начинает выполнять некоторые действия с ним, но не завершает операции с ресурсом до конца. В этот момент может произойти:

- Переключение контекста задачи, то есть процессор начнет выполнять другую задачу.
- Прерывание действия микроконтроллера, вследствие чего процессор займется выполнением обработчика соответствующего прерывания.

Если другая задача или обработчик возникшего прерывания обратятся к этому же самому ресурсу, состояние которого носит промежуточный характер из-за воздействия первой задачи, то результат работы программы будет отличаться от ожидаемого. Рассмотрим несколько примеров.

Доступ к внешней периферии

Рассмотрим сценарий, когда две задачи — задача А и задача Б — выводят информацию на ЖКИ-дисплей. Задача А ответственна за вывод значения каких-либо параметров на дисплей. Задача Б отвечает за вывод экстренных сообщений об авариях:

- Выполняется задача А и начинает выводить очередной параметр на дисплей: «Температура = 25 °С».
- Задача А вытесняется задачей Б в момент, когда на дисплей выведено лишь «Темпе».
- Задача Б выводит на дисплей экстренное сообщение «Превышено давление!!!», после чего переходит в блокированное состояние.
- Задача А возобновляет свое выполнение и выводит оставшуюся часть сообщения на дисплей: «ратура = 25 °С».

В итоге на дисплее появится искаженное сообщение: «ТемпеПревышено давление!!! ратура = 25 °С».

Неатомарные операции чтение/модификация/запись

Пусть стоит задача установить (сбросить, инвертировать — не имеет значения) один бит в регистре специальных функций, в данном случае — в регистре порта ввода/вывода микроконтроллера. Рассмотрим пример кода на языке Си и полученную в результате компиляции последовательность инструкций ассемблера.

Для микроконтроллеров AVR:

```
/* Код на Си */
PORTG ^= (1 << PG3);
/* Скомпилированный машинный код и инструкции ассемблера */
544:  80 91 65 00   lds   r24, 0x0065   ; Загрузить PORTG в регистр общего назначения
548:  98 e0         ldi   r25, 0x08     ; Бит PG3 — в другой регистр
54a:  89 27         eor   r24, r25      ; Операция Исключающее ИЛИ
54c:  80 93 65 00   sts   0x0065, r24   ; Результат — обратно в PORTG
```

Для микроконтроллеров ARM7:

```
/* Код на Си. */
PORTA |= 0x01;
/* Скомпилированный машинный код и инструкции ассемблера */
0x00000264 481C LDR R0,[PC,#0x0070]   ; Получить адрес PORTA
0x00000266 6801 LDR R1,[R0,#0x00]     ; Считать значение PORTA в R1
0x00000268 2201 MOV R2,#0x01          ; Поместить 1 в R2
0x0000026A 4311 ORR R1,R              ; Лог. И регистра R1 (PORTA) и R2 (константа 1)
0x0000026C 6001 STR R1,[R0,#0x00]     ; Сохранить новое значение в PORTA
```

И в первом, и во втором случае последовательность действий сводится:

- к копированию значения порта микроконтроллера в регистр общего назначения,
- к модификации регистра общего назначения,
- к обратному копированию результата из регистра общего назначения в порт.

Такую последовательность действий называют операцией чтения/модификации/записи.

Теперь рассмотрим случай, когда сразу две задачи выполняют операцию чтения/модификации/записи одного и того же порта.
1) Задача А загружает значение порта в регистр.

2) В этот момент ее вытесняет задача Б, при этом задача А не «успела» модифицировать и записать данные обратно в порт.
3) Задача Б изменяет значение порта и, например, блокируется.
4) Задача А продолжает выполняться с точки, в которой ее выполнение было прервано. При этом она продолжает работать с копией порта в регистре, выполняет какие-то действия над ним и записывает значение регистра обратно в порт.

Можно видеть, что в этом случае результат воздействия задачи Б на порт окажется потерян и порт будет содержать неверное значение.

О подобных операциях чтение/модификация/запись говорят, что они не являются атомарными. Атомарными же операциями называют те, выполнение которых не может быть прервано планировщиком. Приводя пример из архитектуры AVR, можно назвать инструкции процессора cbi и sbi, позволяющие сбросить/установить бит в регистре специальных функций. Разумеется, операция длиной в одну машинную инструкцию не может быть прервана планировщиком, то есть является атомарной.

Неатомарными могут быть не только операции с регистрами специальных функций. Операция над любой переменной языка Си, физический размер которой превышает разрядность микроконтроллера, является неатомарной. Например, операция инкремента глобальной переменной типа unsigned long на 8-битной архитектуре AVR выглядит так:

```
/* Код на Си */
unsigned long counter = 0;
counter++;
/* Скомпилированный машинный код и инструкции ассемблера */
618:   80 91 13 01      lds     r24, 0x0113
61c:   90 91 14 01      lds     r25, 0x0114
620:   a0 91 15 01      lds     r26, 0x0115
624:   b0 91 16 01      lds     r27, 0x0116
628:   01 96      adiw  r24, 0x01   ; 1
62a:   a1 1d      adc   r26, r1
62c:   b1 1d      adc   r27, r1
62e:   80 93 13 01      sts     0x0113, r24
632:   90 93 14 01      sts     0x0114, r25
636:   a0 93 15 01      sts     0x0115, r26
63a:   b0 93 16 01      sts     0x0116, r27
```

Если другая задача или прерывание обратятся к этой же переменной в течение этих 11 инструкций, результат окажется искаженным.

Следует отметить, что неатомарными являются также операции с составными типами — структурами, когда модифицируется сразу несколько членов структуры.

Реентерабельность функций

Функция называется реентерабельной, если она корректно работает при одновременном ее вызове из нескольких задач и/или прерываний. Под одновременным вызовом понимается вызов функции из одной задачи в тот момент, когда та уже вызвана из другой задачи, но еще не выполнена до конца.

Во FreeRTOS каждая задача имеет свой собственный стек и свой набор значений регистров процессора. Если функция использует переменные, расположенные только в стеке или в регистрах процессора, то она является реентерабельной. Напротив, функция, которая сохраняет свое состояние между вызовами в статической или глобальной переменной, не является реентерабельной.

Таким образом, функция, которая зависит только от своих параметров, не использует глобальные и статические переменные и вызывает только реентерабельные функции, будет реентерабельной [4].

Одновременный вызов нереентерабельной функции из нескольких задач может привести к непредсказуемому результату. Реентерабельными функциями можно пользоваться, не опасаясь одновременного их вызова из нескольких задач.

Рассмотрим пример реентерабельной функции:

```
/* Параметр передается в функцию через регистр общего назначения или стек. Это безопасно, т. к. каждая задача имеет свой набор регистров и свой стек. */
long lAddOneHundered( long lVar1 )
{
/* Объявлена локальная переменная. Компилятор расположит ее или в регистре или в стеке в зависимости от уровня оптимизации. Каждая задача и каждое прерывание, вызывающее эту функцию, будет иметь свою копию этой локальной переменной. */
long lVar2;
/* Какие-то действия над аргументом и локальной переменной. */
lVar2 = lVar1 + 100;
/* Обычно возвращаемое значение также помещается либо в стек, либо в регистр. */
return lVar2;
}
```

Теперь рассмотрим несколько нереентерабельных функций:

```
/* В этом случае объявлена глобальная переменная. Каждая задача, вызывающая функцию, которая использует эту переменную, будет «иметь дело» с одной и той же копией этой переменной */
long lVar1;

/* Нереентерабельная функция 1 */
long lNonReentrantFunction1( void )
{
/* Какие-то действия с глобальной переменной. */
lVar1 += 10;

return lVar1;
}

/* Нереентерабельная функция 2 */
void lNonReentrantFunction2( void )
{
/* Переменная, объявленная как статическая. Компилятор расположит ее не в стеке. Значит, каждая задача, вызывающая эту функцию, будет «иметь дело» с одной и той же копией этой переменной. */
static long lState = 0;
        switch( lState ) { /* … */};
}

/* Нереентерабельная функция 3 */
long lNonReentrantFunction3( void )
{
/* Функция, которая вызывает нереентерабельную функцию, также является нереентерабельной. */
return lNonReentrantFunction1() + 100;
}
```

Механизм взаимного исключения

Доступ к ресурсу, операции с которым одновременно выполняют несколько задач и/или прерываний, должен контролироваться механизмом взаимного исключения (mutual exclusion).

Механизм взаимного исключения гарантирует, что если задача начала выполнять некоторые действия с ресурсом, то никакая другая задача (или прерывание) не сможет получить доступ к данному ресурсу, пока операции с ним не будут завершены первой задачей.

FreeRTOS предлагает несколько способов реализации механизма взаимного исключения:

- критические секции;
- мьютексы;
- задачи-сторожа.

Однако наилучшая реализация взаимного исключения — это написание программы, в которой ресурсы не разделяются между несколькими задачами и доступ к одному ресурсу выполняет единственная задача или прерывание.

Критические секции

Сразу следует отметить, что критические секции — это очень грубый способ реализации взаимного исключения.

Критическая секция — это часть программы, которую в один момент времени может выполнять только одна задача или прерывание. Обычно защищаемый критической секцией участок кода начинается с инструкции входа в критическую секцию и заканчивается инструкцией выхода из нее.

Во FreeRTOS, в отличие от более сложных операционных систем, существует одна глобальная критическая секция. Если одна задача вошла в критическую секцию, то никакая другая задача не будет выполняться, пока не произойдет выход из критической секции.

FreeRTOS допускает два способа реализации критической секции:

- запрет прерываний;
- приостановка планировщика.

Запрет прерываний

Во FreeRTOS вход в критическую секцию, реализованную запретом прерываний, сводится к запрету всех прерываний процессора или (в зависимости от конкретного порта FreeRTOS) к запрету прерываний с приоритетом равным и ниже макроопределения ***configMAX_SYSCALL_INTERRUPT_PRIORITY***.

Во FreeRTOS участок кода, защищаемый критической секцией, которая реализована запретом прерываний, — это участок кода, окруженный вызовом API-макросов: ***taskENTER_CRITICAL()*** — вход в критическую секцию и ***taskEXIT_CRITICAL()*** — выход из критической секции.

Переключение контекста при вытесняющей многозадачности происходит по прерыванию (обычно от таймера), поэтому задача, которая вызвала ***taskENTER_CRITICAL()***, будет оставаться в состоянии выполнения, пока не вызовет ***taskEXIT_CRITICAL()***.

Участки кода, находящиеся внутри критической секции, должны быть как можно короче и выполняться как можно быстрее. Иначе использование критических секций негативно скажется на времени реакции системы на прерывания.

FreeRTOS допускает вложенный вызов макросов ***taskENTER_CRITICAL()*** и ***taskEXIT_CRITICAL()***, их реализация позволяет сохранять глубину вложенности. Выход программы из критической секции происходит, только если глубина вложенности станет равной нулю. Каждому вызову ***taskENTER_CRITICAL()*** должен соответствовать вызов ***taskEXIT_CRITICAL()***.

Пример использования критической секции:

```
/* Чтобы доступ к порту PORTA не был прерван никакой другой задачей, входим в критическую секцию. */
taskENTER_CRITICAL();
/* Переключение на другую задачу не может произойти, когда выполняется код, окруженный вызовом taskENTER_CRITICAL() и taskEXIT_CRITICAL().
Прерывания здесь могут происходить, только если микроконтроллер допускает вложение прерываний. Прерывание выполнится, если его приоритет выше константы configMAX_SYSCALL_INTERRUPT_PRIORITY. Однако такие прерывания не могут вызывать FreeRTOS API-функции. */
PORTA |= 0x01;
/* Неатомарная операция чтение/модификация/запись завершена. Сразу после этого выходим из критической секции. */
taskEXIT_CRITICAL();
```

Рассматривая пример выше, следует отметить, что если внутри критической секции произойдет прерывание с приоритетом выше ***configMAX_SYSCALL_INTERRUPT_PRIORITY***, которое, в свою очередь, обратится к порту PORTA, то принцип взаимного исключения доступа к ресурсу будет нарушен.

Приостановка/запуск планировщика

Еще один способ реализации критической секции в FreeRTOS — это приостановка работы планировщика (suspending the scheduler).

В отличие от реализации критической секции с помощью запрета прерываний (макросы ***taskENTER_CRITICAL()*** и ***taskEXIT_CRITICAL()***), которые защищают участок кода от доступа как из задач, так и из прерываний, реализация с помощью приостановки планировщика защищает участок кода только от доступа из другой задачи. Все прерывания микроконтроллера остаются разрешены.

Операция запуска планировщика после приостановки выполняется существенно дольше макроса ***taskEXIT_CRITICAL()***, это немаловажно с точки зрения сокращения времени выполнения критических секций в программе. Этот момент следует учитывать при выборе способа организации критических секций.

Приостановка планировщика выполняется API-функцией ***vTaskSuspendAll()***. Ее прототип:

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

После вызова ***vTaskSuspendAll()*** планировщик останавливается, переключения контекста каждый системный квант времени не происходит, задача, которая вызвала ***vTaskSuspendAll()***, будет выполняться сколь угодно долго до запуска планировщика. API-функция ***vTaskSuspendAll()*** не влияет на прерывания: если до вызова ***vTaskSuspendAll()*** они были разрешены, то при возникновении прерываний их обработчики будут выполняться.

Если же обработчик прерывания выполнил макрос принудительного переключения контекста (***portSWITCH_CONTEXT()***, ***taskYIELD()***, ***portYIELD_FROM_ISR()*** и др.— в зависимости от порта FreeRTOS), то запрос на переключение контекста будет выполнен, как только работа планировщика будет возобновлена.

Другие API-функции FreeRTOS нельзя вызывать, когда планировщик приостановлен вызовом ***vTaskSuspendAll()***.

Для возобновления работы планировщика служит API-функция ***xTaskResumeAll()***, ее прототип:

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

Возвращаемое значение может быть равно:

- ***pdTRUE*** — означает, что переключение контекста произошло сразу после возобновления работы планировщика.
- ***pdFALSE*** — во всех остальных случаях.

Возможен вложенный вызов ***API-функций vTaskSuspendAll()*** и ***xTaskResumeAll()***. При этом ядро автоматически подсчитывает глубину вложенности. Работа планировщика будет возобновлена, если глубина вложенности станет равна 0. Этого можно достичь, если каждому вызову ***vTaskSuspendAll()*** будет соответствовать вызов ***xTaskResumeAll()***.

Мьютексы

Взаимное исключение называют также мьютексом (mutex — MUTual EXclusion), этот термин чаще используется в операционных системах Windows и Unix-подобных [5].

Мьютекс во FreeRTOS представляет собой специальный тип двоичного семафора, который используется для реализации совместного доступа к ресурсу двух или большего числа задач. При использовании в качестве механизма взаимного исключения мьютекс можно представить как семафор, относящийся к ресурсу, доступом к которому необходимо управлять.

В отличие от семафора мьютекс во FreeRTOS предоставляет механизм наследования приоритетов, о котором будет рассказано ниже. Также следует отметить, что использование мьютекса из тела обработчика прерывания невозможно.

Чтобы корректно получить доступ к ресурсу, задача должна предварительно захватить мьютекс, стать его владельцем. Когда владелец семафора закончил операции с ресурсом, он должен отдать мьютекс обратно. Только когда мьютекс освободился (возвращен какой-либо задачей), другая задача может его захватить и безопасно выполнить свои операции с общим для нескольких задач ресурсом. Задаче не разрешено выполнять операции с ресурсом, если в данный момент она не является владельцем мьютекса. Процессы, происходящие при взаимном исключении доступа с использованием мьютекса, приведены на рис. 1.

Обе задачи нуждаются в доступе к ресурсу, однако только задача-владелец мьютекса может его получить (рис. 1а). Задача А пытается захватить мьютекс, в этот момент он свободен, поэтому она становится его владельцем (рис. 1б). Задача А выполняет некоторые действия с ресурсом. В этот момент задача Б пытается захватить тот же самый мьютекс, однако это ей не удается, потому что задача А все еще является его владельцем. Соответственно, пока задача А выполняет операции с ресурсом, задача Б не может получить к нему доступ и переходит в блокированное состояние (рис. 1в). Задача А до конца завершает операции с ресурсом и возвращает мьютекс обратно (рис. 1г). Это приводит к разблокировке задачи Б, теперь она получает доступ к ресурсу (рис. 1д). При завершении действий с ресурсом задача Б обязана отдать мьютекс обратно (рис. 1е).

Легко заметить, что мьютексы и двоичные семафоры очень похожи в использовании. Отличие заключается в том, что мьютекс после захвата обязательно должен быть возвращен, иначе другие задачи не смогут получить доступ к разделяемому ресурсу. Двоичный семафор, используемый в целях синхронизации выполнения задач (и прерываний), наоборот — не должен возвращаться задачей, которая его захватила.

Важным моментом является то, что непосредственно мьютекс не защищает ресурс от одновременного доступа нескольких задач. Вместо этого реализация всех задач в системе должна быть выполнена так, чтобы перед инструкцией доступа к ресурсу следовал вызов API-функции захвата соответствующего мьютекса. Эта обязанность ложится на программиста.

Работа с мьютексами

Мьютекс представляет собой специальный вид семафора, поэтому доступ к мьютексу осуществляется так же, как и к семафору: с помощью дескриптора (идентификатора) мьютекса — переменной типа ***xSemaphoreHandle***.

Для того чтобы API-функции для работы с мьютексами были включены в программу, необходимо установить макроопределение ***configUSE_MUTEXES*** в файле ***FreeRTOSConfig.h*** равным «1».

Мьютекс должен быть явно создан перед первым его использованием. API-функция ***xSemaphoreCreateMutex()*** служит для создания мьютекса:

```
xSemaphoreHandle xSemaphoreCreateMutex( void );
```

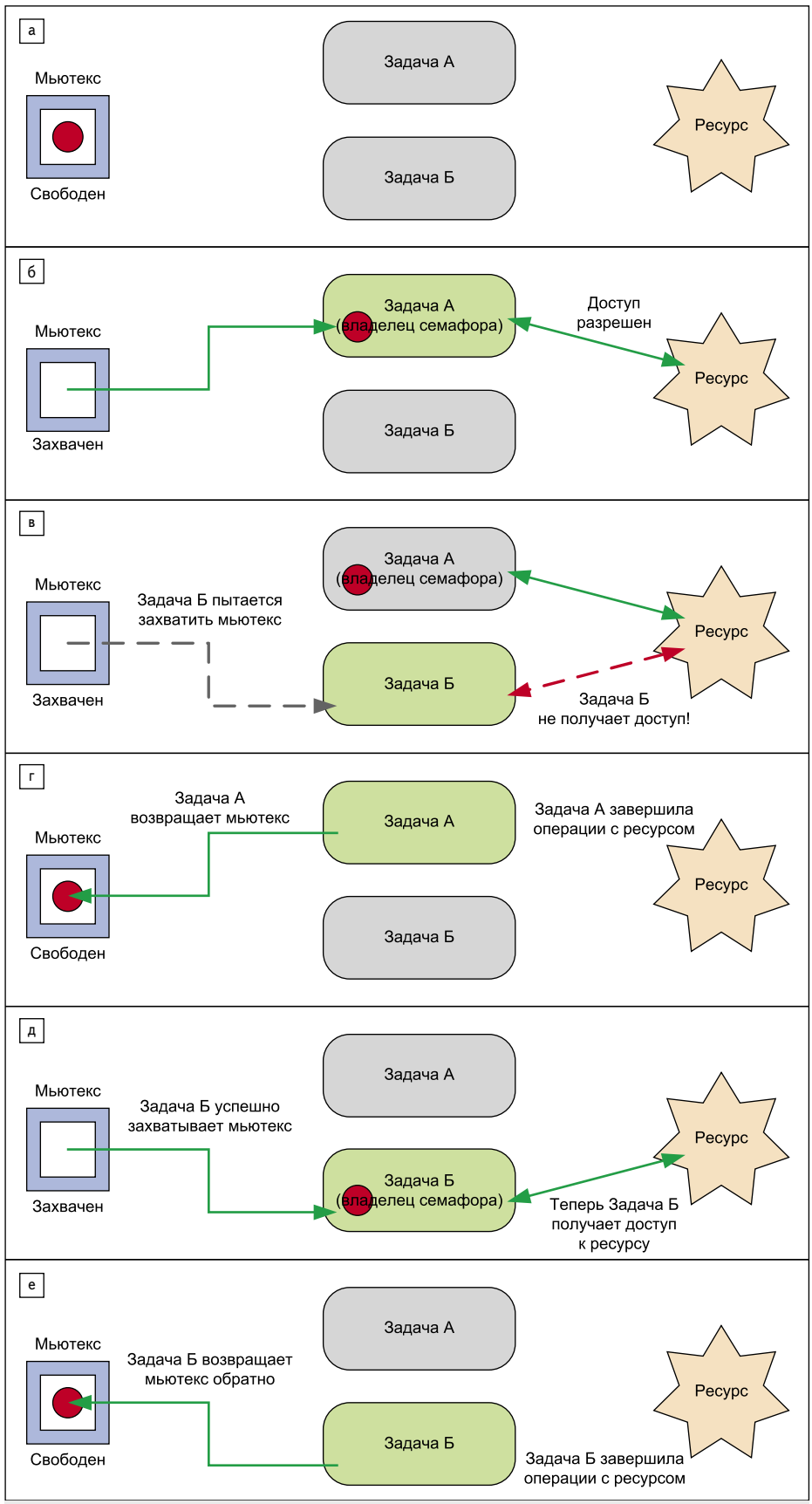

Рис. 1. Использование мьютекса для управления доступом к ресурсу

Возвращаемое значение — дескриптор мьютекса, он должен быть сохранен в переменной для дальнейшего обращения к мьютексу. Если мьютекс не создан по причине отсутствия достаточного объема памяти, возвращаемым значением будет NULL.

Операции захвата и возврата (выдачи) мьютекса выполняются с помощью аналогичных API-функций для работы с семафорами — ***xSemaphoreTake()*** и ***xSemaphoreGive()***, которые были рассмотрены в [1, № 7].

Рассмотрим, как применение мьютекса позволяет решить проблему совместного доступа к ресурсу, на примере учебной программы № 1. В качестве разделяемого ресурса выступает консоль, две задачи выводят свое сообщение на дисплей. Обратите внимание на реализацию вывода строки на консоль: вместо стандартной функции используется посимвольный вывод.

Сначала рассмотрим учебную программу № 1 без использования мьютекса:

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

/* Дескриптор мьютекса — глобальная переменная*/
volatile xSemaphoreHandle xMutex;

/* Функция посимвольно выводит строку на консоль.
Консоль, как ресурс, никаким образом не защищена от совместного
доступа из нескольких задач. */
static void prvNewPrintString(const portCHAR *pcString) {
  portCHAR *p;
  int i;

  /* Указатель — на начало строки */
  p = pcString;
  /* Пока не дошли до нулевого символа — конца строки. */
  while (*p) {
    /* Вывод на консоль символа, на который ссылается указатель. */
    putchar(*p);
    /* Указатель — на следующий символ в строке. */
    p++;
    /* Вывести содержимое буфера экрана на экран. */
    fflush(stdout);
    /* Небольшая пауза */
    for (i = 0; i < 10000; i++) ;
  }
}

/* Функция, реализующая задачу.
 Будет создано 2 экземпляра этой задачи.
 Каждый получит строку символов в качестве аргумента
 при создании задачи. */
static void prvPrintTask(void *pvParameters) {
    char *pcStringToPrint;
    pcStringToPrint = (char *) pvParameters;
    for (;;) {
      /* Для вывода строки на консоль используется своя
         функция prvNewPrintString(). */
      prvNewPrintString(pcStringToPrint);
      /* Блокировать задачу на промежуток времени случайной
         длины: от 0 до 500 мс. */
      vTaskDelay((rand() % 500));
      /* Вообще функция rand() не является реентерабельной.
         Однако в этой программе это неважно. */
    }
}

/*-----------------------------------------------------------*/
/* Точка входа. С функции main() начнется выполнение
   программы. */
short main( void )
{
    /* Создание мьютекса. */
    xMutex = xSemaphoreCreateMutex();
    /* Создание задач, если мьютекс был успешно создан. */
    if (xMutex != NULL) {
      /* Создать два экземпляра одной задачи. Каждому
         экземпляру задачи передать в качестве аргумента свою
         строку. Приоритет задач задать разным, чтобы имело
         место вытеснение задачи 1 задачей 2.
      */ xTaskCreate(prvPrintTask, "Print1", 1000,
            "Task 1 ******************************************\r\n", 1,
          NULL);
        xTaskCreate(prvPrintTask, "Print2", 1000,
            "Task 2 ------------------------------------\r\n", 2,
          NULL);
          /* Запуск планировщика. */
          vTaskStartScheduler();
    }
    return 1;
}
```

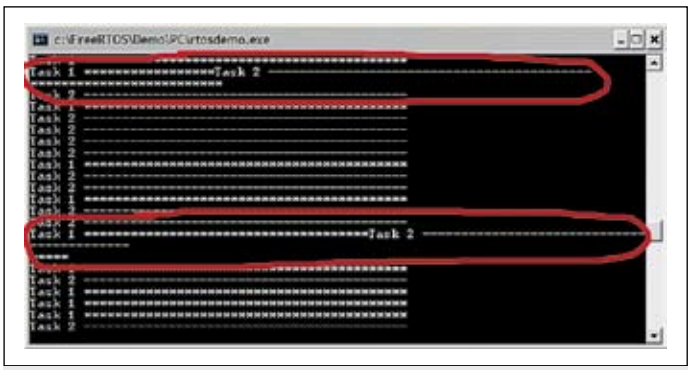

Рис. 2. Результат работы учебной программы № 1 без использования мьютекса

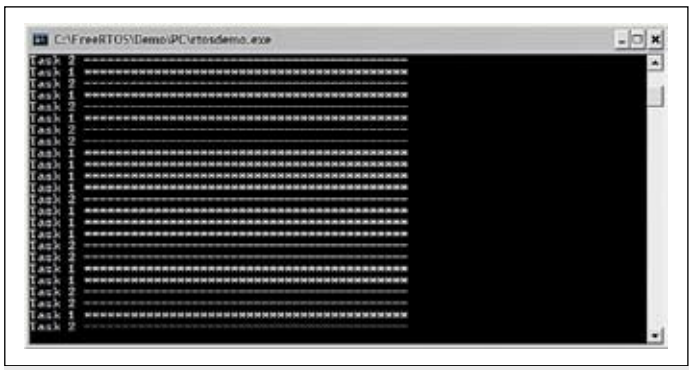

Рис. 3. Результат работы учебной программы № 1 с применением мьютекса

Как видно по результатам работы (рис. 2), совместный доступ к консоли без применения какого-либо механизма взаимного исключения приводит к тому, что некоторые сообщения, которые выводят на консоль задачи, оказываются повреждены.

Теперь защитим консоль от одновременного доступа с помощью мьютекса, заменив реализацию функции ***prvNewPrintString()*** на следующую:

```
/* Функция посимвольно выводит строку на консоль.
Консоль, как ресурс, защищена от совместного доступа
из нескольких задач с помощью мьютекса. */
static void prvNewPrintString(const portCHAR *pcString) {
   portCHAR *p;
   int i;

   /* Указатель — на начало строки */
   p = pcString;
   /* Захватить мьютекс. Время ожидания в блокированном
    состоянии, если мьютекс недоступен, сколь угодно долго.
    Возвращаемое значение xSemaphoreTake() должно проверяться,
    если указано время пребывания в блокированном состоянии,
    отличное от portMAX_DELAY */
   xSemaphoreTake( xMutex, portMAX_DELAY ); {
      /* Пока не дошли до нулевого символа — конца строки. */
      while (*p) {
         /* Вывод на консоль символа, на который ссылается указатель. */
         putchar(*p);
         /* Указатель — на следующий символ в строке. */
         p++;
         /* Вывести содержимое буфера экрана на экран. */
         fflush(stdout);
         /* Небольшая пауза */
         for (i = 0; i < 10000; i++) ;
      }
   }
   /* Когда вывод ВСЕЙ строки на консоль закончен,
    освободить мьютекс. Иначе другие задачи не смогут
    обратиться к консоли! */
   xSemaphoreGive( xMutex );
}
```

Теперь при выполнении учебной программы № 1 сообщения от разных задач не накладываются друг на друга: совместный доступ к ресурсу (консоли) организован правильно (рис. 3).

Рекурсивные мьютексы

Помимо обычных мьютексов, рассмотренных выше, FreeRTOS поддерживает также рекурсивные мьютексы (Recursive Mutexes) [6]. Их основное отличие от обычных мьютексов заключается в том, что они корректно работают при вложенных операциях захвата и освобождения мьютекса. Вложенные операции захвата/освобождения мьютекса допускаются только в теле задачи-владельца мьютекса. Рассмотрим пример рекурсивного захвата обычного мьютекса:

```
xSemaphoreHandle xMutex;
/* ... */
xMutex = xSemaphoreCreateMutex();
/* ... */

/* Функция, реализующая задачу. */
void vTask(void *pvParameters) {
   for (;;) {
      /* Захват мьютекса */
      xSemaphoreTake( xMutex, portMAX_DELAY );
      /* Действия с ресурсом */
      /* ... */
      /* Вызов функции, которая выполняет операции с этим же
         ресурсом. */
      vSomeFunction();
      /* Действия с ресурсом */
      /* ... */
      /* Действия с ресурсом закончены. Освободить мьютекс. */
      xSemaphoreGive( xMutex );
   }
}

/* Функция, которая вызывается из тела задачи vTask*/
void vSomeFunction(void) {
   /* Захватить тот же самый мьютекс.
      Т. к. тайм-аут не указан, то задача «зависнет» в ожидании,
      пока мьютекс не освободится.
      Однако это никогда не произойдет! */
   if (xSemaphoreTake( xMutex, portMAX_DELAY ) == pdTRUE )
{
      /* Действия с ресурсом внутри функции */
      /* ... */
      /* Освободить мьютекс */
      xSemaphoreGive( xMutex );
   }
}
```

Такое использование обычного мьютекса приведет к краху программы. При попытке повторно захватить мьютекс внутри функции ***vSomeFunction()*** задача ***vTask*** перейдет в блокированное состояние, пока мьютекс не будет возвращен. Другие задачи смогут выполнить возврат мьютекса только после того, как сами его захватят. Однако мьютекс уже захвачен, поэтому задача ***vTask*** заблокируется на бесконечно долгое время («зависнет»).

Если при повторном вызове API-функции ***xSemaphoreTake()*** было указано конечное время тайм-аута, «зависания» задачи не произойдет. Вместо этого действия с ресурсом, выполняемые внутри функции, никогда не будут произведены, что также является недопустимой ситуацией.

Когда программа проектируется так, что операции захват/освобождение мьютекса являются вложенными, следует использовать рекурсивные мьютексы:

```
xSemaphoreHandle xRecursiveMutex;
/* ... */
xRecursiveMutex = xSemaphoreCreateRecursiveMutex();
/* ... */

/* Функция, реализующая задачу. */
void vTask(void *pvParameters) {
   for (;;) {
      /* Захват мьютекса */
      xSemaphoreTakeRecursive( xRecursiveMutex, portMAX_DELAY );
      /* Действия с ресурсом */
      /* ... */
      /* Вызов функции, которая выполняет операции с этим же
         ресурсом. */
      vSomeFunction();
      /* Действия с ресурсом */
      /* ... */
      /* Действия с ресурсом закончены. Освободить мьютекс. */
      xSemaphoreGiveRecursive( xRecursiveMutex );
   }
}

/* Функция, которая вызывается из тела задачи vTask*/
void vSomeFunction(void) {
   /* Захватить тот же самый мьютекс.
      При этом состояние мьютекса никоим образом не изменится.
      Задача не заблокируется, действия с ресурсом внутри этой
      функции будут выполнены. */
   if (xSemaphoreTakeRecursive( xRecursiveMutex, portMAX_DELAY
) == pdTRUE ) {
      /* Действия с ресурсом внутри функции */
      /* ... */
      /* Освободить мьютекс */
      xSemaphoreGiveRecursive( xRecursiveMutex );
   }
}
```

В этом случае программа будет работать корректно. При повторном захвате мьютекса API-функцией ***xSemaphoreTakeRecursive()*** задача не перейдет в блокированное состояние, и эта же задача останется владельцем мьютекса. Вместо этого увеличится на единицу внутренний счетчик, который определяет, сколько операций «захват» было применено к мьютексу, действия с ресурсом внутри функции ***vSomeFunction()*** будут выполнены, так как задача ***vTask*** остается владельцем мьютекса. При освобождении мьютекса (при вызове API-функции ***xSemaphoreGiveRecursive()***) из тела одной и той же задачи внутренний счетчик уменьшается на единицу. Когда этот счетчик станет равен нулю, это будет означать, что текущая задача больше не является владельцем мьютекса и теперь он может быть захвачен другой задачей.

Таким образом, каждому вызову API-функции ***xSemaphoreTakeRecursive()*** внутри тела одной и той же задачи должен соответствовать вызов API-функции ***xSemaphoreGiveRecursive()***.

Для того чтобы использовать рекурсивные мьютексы в программе, необходимо установить макроопределение ***configUSE_RECURSIVE_MUTEXES*** в файле ***FreeRTOSConfig.h*** равным «1».

Как и обращение к обычному мьютексу, обращение к рекурсивному мьютексу осуществляется с помощью дескриптора (идентификатора) мьютекса — переменной типа ***xSemaphoreHandle***.

API-функции для работы с рекурсивными мьютексами:

- ***xSemaphoreCreateRecursiveMutex()*** — создание рекурсивного мьютекса;
- ***xSemaphoreTakeRecursive()*** — захват рекурсивного мьютекса;
- ***xSemaphoreGiveRecursive()*** — освобождение (возврат) рекурсивного мьютекса.

Набор параметров и возвращаемое значение этих API-функций ничем не отличаются от соответствующих API-функций для работы с обычными мьютексами. Стоит помнить лишь о том, что API-функции для работы с рекурсивными мьютексами нельзя применять к обычным мьютексам и наоборот.

Проблемы при использовании мьютексов

Инверсия приоритетов

Вернемся к рассмотрению учебной программы № 1. Возможная последовательность выполнения задач приведена на рис. 4. Такая последовательность имела бы место, если во FreeRTOS не был бы реализован механизм наследования приоритетов, о котором будет рассказано ниже.

Пусть в момент времени (1) низкоприоритетная задача 1 вытеснила задачу Бездействие, так как закончился период пребывания задачи 1 в блокированном состоянии (рис. 4). Задача 1 захватывает мьютекс (становится его владельцем) и начинает посимвольно выводить свою строку на дисплей (2). В момент времени (3) разблокируется высокоприоритетная задача 2, при этом она вытесняет задачу 1, когда та еще не закончила вывод строки на дисплей. Задача 2 пытается захватить мьютекс, однако он уже захвачен задачей 1, поэтому задача 2 блокируется в ожидании, когда мьютекс станет доступен. Управление снова получает задача 1, она завершает вывод строки на дисплей — операция с ресурсом завершена. Задача 1 возвращает мьютекс обратно — мьютекс становится доступен (момент времени (4)). Как только мьютекс становится доступен, разблокируется задача 2, которая ожидала его освобождения. Задача 2 захватывает мьютекс (становится его владельцем) и выводит свою строку на дисплей. Приоритет задачи 2 выше, чем у задачи 1, поэтому задача 2 выполняется все время, пока полностью не выведет свою строку на дисплей, после чего она отдает мьютекс обратно и блокируется на заданное API-функцией vTaskDelay() время — момент времени (5). Задача 1 снова получает управление, но на непродолжительное время — пока также не перейдет в блокированное состояние, вызвав vTaskDelay().

Учебная программа № 1 и рис. 4 демонстрируют одну из возможных проблем, возникающих при использовании мьютексов для реализации механизма взаимного исключения, — проблему инверсии приоритетов (Priority Inversion) [7]. На рис. 4 представлена ситуация, когда высокоприоритетная задача 2 вынуждена ожидать, пока низкоприоритетная задача 1 завершит действия с ресурсом и возвратит мьютекс обратно. То есть на некоторое время фактический приоритет задачи 2 оказывается ниже приоритета задачи 1: происходит инверсия приоритетов.

В реальных программах инверсия приоритетов может оказывать еще более негативное влияние на выполнение высокоприоритетных задач. Рассмотрим пример. В программе могут существовать также задачи со «средним» приоритетом — ниже, чем у высокоприоритетной, которая ожидает освобождения мьютекса, но выше, чем у низкоприоритетной, которая в данный момент захватила мьютекс и выполняет действия с разделяемым ресурсом. Среднеприоритетные задачи могут разблокироваться на протяжении интервала, когда низкоприоритетная задача владеет мьютексом. Такой сценарий является

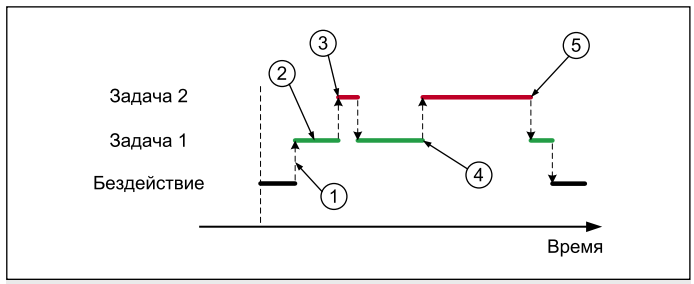

Рис. 4. Переключение между задачами в учебной программе № 1 без механизма наследования приоритетов

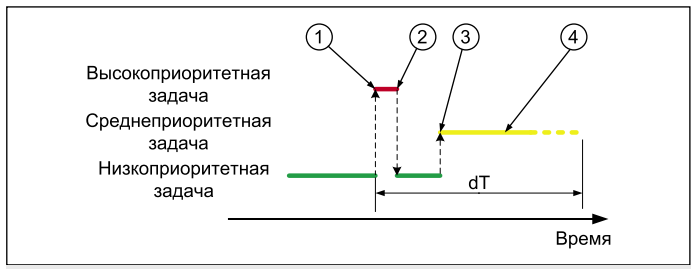

Рис. 5. Наихудший случай влияния инверсии приоритетов

наихудшим, так как ко времени, когда высокоприоритетная задача ожидает освобождения мьютекса, будет добавлено время выполнения среднеприоритетных задач (рис. 5).

Низкоприоритетная задача стала владельцем мьютекса ранее. Происходит некоторое событие, за обработку которого отвечает высокоприоритетная задача. Она разблокируется и пытается захватить мьютекс (1), это ей не удается, и она блокируется — момент времени (2) на рис. 5. Управление снова возвращается низкоприоритетной задаче, которая в момент времени (3) вытесняется задачей, приоритет которой выше (среднеприоритетной задачей). Среднеприоритетная задача может выполняться продолжительное время, в течение которого высокоприоритетная будет ожидать, пока мьютекс не будет освобожден низкоприоритетной задачей (4). Время реакции на событие при этом значительно удлиняется — величина dT на рис. 5.

В итоге инверсия приоритетов может значительно ухудшить время реакции микроконтроллерной системы на внешние события.

Для уменьшения (но не полного исключения) негативного влияния инверсии приоритетов во FreeRTOS реализован механизм наследования приоритетов (Priority Inheritance). Его работа заключается во временном увеличении приоритета низкоприоритетной задачи-владельца мьютекса до уровня приоритета высокоприоритетной задачи, которая в данный момент пытается захватить мьютекс. Когда низкоприоритетная задача освобождает мьютекс, ее приоритет уменьшается до значения, которое было до повышения. Говорят, что низкоприоритетная задача наследует приоритет высокоприоритетной задачи.

Рассмотрим работу механизма наследования приоритетов на примере программы с высоко-, средне- и низкоприоритетной задачами (рис. 6).

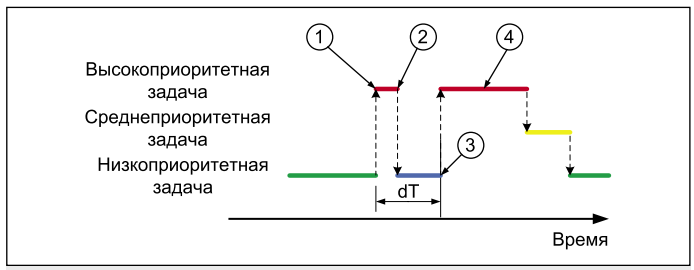

Рис. 6. Уменьшение влияния инверсии приоритетов при работе механизма наследования приоритетов

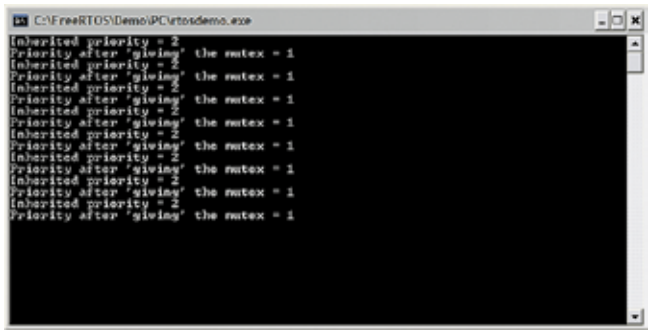

Рис. 7. Результат работы учебной программы № 2

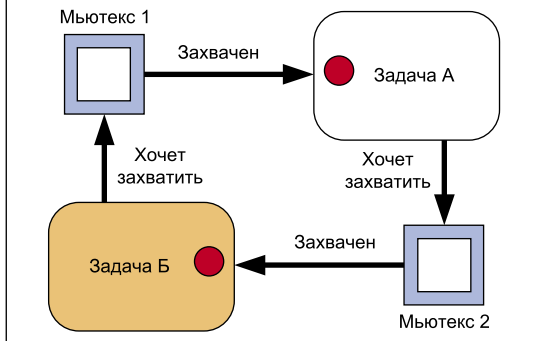

Рис. 8. Взаимная блокировка двух задач

Низкоприоритетная задача стала владельцем мьютекса ранее. Происходит некоторое событие, за обработку которого отвечает высокоприоритетная задача. Она разблокируется и пытается захватить мьютекс (1), это ей не удается, и она блокируется — момент времени (2) на рис. 6. Однако в результате попытки высокоприоритетной задачи захватить мьютекс низкоприоритетная задача-владелец мьютекса наследует приоритет этой высокоприоритетной задачи. Теперь низкоприоритетная задача не может быть вытеснена среднеприоритетной задачей. Поэтому в момент времени (3), когда низкоприоритетная задача завершила операции с ресурсом и возвращает мьютекс, разблокируется, захватывает мьютекс и начинает выполняться высокоприоритетная задача (4). Приоритет же низкоприоритетной задачи при этом возвращается к своему «нормальному» значению.

Таким образом, механизм наследования приоритетов уменьшает время реакции системы на событие, когда происходит инверсия приоритетов (сравните величину dT на рис. 5 и рис. 6).

Продемонстрировать работу механизма наследования приоритетов во FreeRTOS позволяет учебная программа № 2. В программе выполняются две задачи: низкоприоритетная задача 1 с приоритетом 1 и высокоприоритетная задача 2 с приоритетом 2. Обе задачи пытаются захватить один и тот же мьютекс. Низкоприоритетная задача сигнализирует на дисплей, если ее приоритет изменился (повысился):

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

/* Дескриптор мьютекса — глобальная переменная*/
volatile xSemaphoreHandle xMutex;

/* Низкоприоритетная задача 1. Приоритет = 1. */
static void prvTask1(void *pvParameters) {
  long i;
  /* Логическая переменная. Определяет, произошло ли
     наследование приоритетов. */
  unsigned portBASE_TYPE uxIsPriorityInherited = pdFALSE;
  /* Бесконечный цикл */
  for (;;) {
    /* Наследования приоритетов еще не было */
    uxIsPriorityInherited = pdFALSE;
    /* Захватить мьютекс. */
    xSemaphoreTake(xMutex, portMAX_DELAY);
    /* Какие-то действия. За это время высокоприоритетная
       задача попытается захватить мьютекс. */
    for (i = 0; i < 100000L; i++)
      ;
    /* Если приоритет этой задачи изменился (был унаследован
       от задачи 2). */
    if (uxTaskPriorityGet(NULL) != 1) {
      printf("Inherited priority = %d\n\r", uxTaskPriorityGet(NULL));
      uxIsPriorityInherited = pdTRUE;
    }
    /* Освободить мьютекс. */
    xSemaphoreGive(xMutex);
    /* Вывести значение приоритета ПОСЛЕ освобождения
       мьютекса. */
    if (uxIsPriorityInherited == pdTRUE) {
      printf("Priority after 'giving' the mutex = %d\n\r",
uxTaskPriorityGet(NULL));
    }
    /* Блокировать задачу на промежуток времени случайной
       длины: от 0 до 500 мс. */
    vTaskDelay((rand() % 500));
  }
}

/* Высокоприоритетная задача 2. Приоритет = 2. */
static void prvTask2(void *pvParameters) {
    for (;;) {
        xSemaphoreTake( xMutex, portMAX_DELAY );
        xSemaphoreGive( xMutex );
        /* Интервал блокировки короче — от 0 до 50 мс */
          vTaskDelay((rand() % 50));
    }
}

/*-----------------------------------------------------------*/
/* Точка входа. С функции main() начнется выполнение программы. */
short main( void )
{
    /* Создание мьютекса. */
    xMutex = xSemaphoreCreateMutex();
    /* Создание задач, если мьютекс был успешно создан. */
    if (xMutex != NULL) {
        xTaskCreate(prvTask1, "prvTask1", 1000, NULL, 1, NULL);
        xTaskCreate(prvTask2, "prvTask2", 1000, NULL, 2, NULL);
        /* Запуск планировщика. */
        vTaskStartScheduler();
    }
    return 1;
}
```

По результатам выполнения учебной программы № 2 (рис. 7) видно, что приоритет задачи 1 временно увеличивается со значения 1 до значения 2, когда задача 2 пытается захватить мьютекс, который уже захвачен задачей 1. После того как задача 1 освобождает мьютекс, ее приоритет возвращается к первоначальному значению.

Следует отметить, что механизм наследования приоритетов во FreeRTOS только уменьшает, однако не устраняет полностью негативное влияние инверсии приоритетов. Поэтому рекомендуется проектировать программу так, чтобы избегать ситуации инверсии приоритетов.

Взаимная блокировка

Взаимная блокировка (Deadlock или Deadly Embrace) — это ситуация в многозадачной системе, когда несколько задач находятся в состоянии бесконечного ожидания доступа к ресурсам, занятым самими этими задачами [8].

Простейший пример взаимной блокировки включает две задачи — задачу А и задачу Б и два мьютекса — мьютекс 1 и мьютекс 2. Взаимная блокировка может произойти при такой последовательности событий:

- Выполняется задача А, которая успешно захватывает мьютекс 1.
- Задача Б вытесняет задачу А.
- Задача Б успешно захватывает мьютекс 2, после чего пытается захватить и мьютекс 1. Это ей не удается, и она блокируется в ожидании освобождения мьютекса 1.
- Управление снова получает задача А. Она пытается захватить мьютекс 2, однако он уже захвачен задачей Б. Поэтому задача А блокируется в ожидании освобождения мьютекса 2.

В итоге получаем ситуацию, когда задача А заблокирована в ожидании освобождения мьютекса 2, захваченного задачей Б. Задача Б заблокирована в ожидании освобождения мьютекса 1, захваченного задачей А. Графически эта ситуация представлена на рис. 8.

Впрочем, в состояние взаимной блокировки может попасть любое количество задач, находящихся в круговой зависимости друг от друга. Если ситуация взаимной блокировки единожды наступила, то выход из этой ситуации невозможен.

Как и в случае с инверсией приоритетов, лучший способ избежать взаимной блокировки задач — это исключить такую возможность на этапе проектирования программы, то есть не создавать круговой зависимости задач друг от друга.

Следует отметить, что помимо рассмотренных выше проблем совместного доступа к ресурсам существуют еще такие, как голодание (Starvation) и разновидность взаимной блокировки, при которой задачи не блокируются, но и не выполняют полезной работы (Livelock). Подробнее с ними можно ознакомиться в [9].

Функция vApplicationTickHook()

Прежде чем продолжить изучение механизмов взаимного исключения, стоит обратить внимание на еще одну возможность FreeRTOS. Как известно, подсистема времени FreeRTOS [1, № 4] основывается на системном кванте времени. По прошествии каждого кванта времени ядро FreeRTOS выполняет внутренние системные действия, связанные как с работой планировщика, так и с отсчетом произвольных временных промежутков.

Программисту предоставляется возможность определить свою функцию, которая будет вызываться каждый системный квант времени. Такая возможность может оказаться полезной, например, для реализации механизма программных таймеров.

Чтобы задать свою функцию, которая будет вызываться каждый системный квант времени, необходимо в файле настроек ядра ***FreeRTOSConfig.h*** задать макроопределение ***configUSE_TICK_HOOK*** равным 1. Сама функция должна содержаться в программе и иметь следующий прототип:

```
void vApplicationTickHook( void );
```

Как и функция задачи Бездействие, функция ***vApplicationTickHook()*** является функцией-ловушкой или функцией обратного вызова (callback function). Поэтому в программе не должны встречаться явные вызовы этой функции.

Отсчет квантов времени во FreeRTOS реализован за счет использования прерывания от одного из аппаратных таймеров микроконтроллера, вследствие чего функция ***vApplicationTickHook()*** вызывается из обработчика прерывания. Поэтому к ней предъявляются следующие требования:

- Она должна выполняться как можно быстрее.
- Должна использовать как можно меньше стека.
- Не должна содержать вызовы API-функций, кроме предназначенных для вызова из обработчика прерывания (то есть чьи имена заканчиваются на ***FromISR*** или ***FROM_ISR***).

Задачи-сторожа (Gatekeeper tasks)

Задачи-сторожа предоставляют простой и прозрачный метод реализации механизма взаимного исключения, которому не присущи проблемы инверсии приоритетов и взаимной блокировки.

Задача-сторож — это задача, которая имеет единоличный доступ к разделяемому ресурсу. Никакая другая задача в программе не имеет права обращаться к ресурсу напрямую. Вместо этого все задачи, разделяющие общий ресурс, обращаются к задаче-сторожу, используя безопасные механизмы межзадачного взаимодействия FreeRTOS. Непосредственно действия с ресурсом выполняет задача-сторож.

В отличие от мьютексов, работать с которыми могут только задачи, к задаче-сторожу могут обращаться как задачи, так и обработчики прерываний.

Рассмотрим использование задачи-сторожа на примере учебной программы № 3. Как и в учебной программе № 1, здесь разделяемым ресурсом выступает консоль. В программе созданы две задачи, каждая из которых выводит свое сообщение на консоль. Кроме того, сообщения выводит функция, вызываемая каждый системный квант времени, это демонстрирует возможность обращения к разделяемому ресурсу из тела обработчика прерывания:

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

/* Прототип задачи, которая выводит сообщения на консоль,
   передавая их задаче-сторожу.
 * Будет создано 2 экземпляра этой задачи */
static void prvPrintTask(void *pvParameters);

/* Прототип задачи-сторожа */
static void prvStdioGatekeeperTask(void *pvParameters);

/* Таблица строк, которые будут выводиться на консоль */
static char *pcStringsToPrint[] = {
   "Task 1 ****************************************************\r\
n",
   "Task 2 ----------------------------------------------------\r\n",
   "Message printed from the tick hook interrupt ##############\r\n"
};

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

/* Объявить очередь, которая будет использоваться для передачи
   сообщений от задач и прерываний к задаче-сторожу. */
xQueueHandle xPrintQueue;

int main(void) {
   /* Создать очередь длиной макс. 5 элементов типа
      "указатель на строку" */
   xPrintQueue = xQueueCreate(5, sizeof(char *));

   /* Проверить, успешно ли создана очередь. */
   if (xPrintQueue != NULL) {
      /* Создать два экземпляра задачи, которые будут выводить
         строки на консоль, передавая их задаче-сторожу.
         В качестве параметра при создании задачи передается
         номер строки в таблице. Задачи создаются с разными
         приоритетами. */
      xTaskCreate(prvPrintTask, "Print1", 1000, (void *) 0, 1, NULL);
      xTaskCreate(prvPrintTask, "Print2", 1000, (void *) 1, 2, NULL);

      /* Создать задачу-сторож. Только она будет иметь
         непосредственный доступ к консоли. */
      xTaskCreate(prvStdioGatekeeperTask, "Gatekeeper", 1000, NULL,
0, NULL);

      /* Запуск планировщика. */
      vTaskStartScheduler();
   }
   return 0;
}
/*-----------------------------------------------------------*/

static void prvStdioGatekeeperTask(void *pvParameters) {
   char *pcMessageToPrint;

   /* Задача-сторож. Только она имеет прямой доступ к консоли.
    * Когда другие задачи "хотят" вывести строку на консоль,
      они записывают указатель на нее в очередь.
    * Указатель из очереди считывает задача-сторож
      и непосредственно выводит строку */
   for (;;) {
      /* Ждать появления сообщения в очереди. */
      xQueueReceive(xPrintQueue, &pcMessageToPrint, portMAX_
DELAY);
      /* Непосредственно вывести строку. */
      printf("%s", pcMessageToPrint);
      fflush(stdout);
      /* Вернуться к ожиданию следующей строки. */
   }
}
/*-----------------------------------------------------------*/

/* Задача, которая автоматически вызывается каждый системный
   квант времени.
 * Макроопределение configUSE_TICK_HOOK должно быть равно 1. */
void vApplicationTickHook(void) {
   static int iCount = 0;
   portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;

   /* Выводить строку каждые 200 квантов времени.
      Строка не выводится напрямую, указатель на нее помещается
      в очередь и считывается задачей-сторожем. */
   iCount++;
   if (iCount >= 200) {
      /* Используется API-функция, предназначенная для вызова
         из обработчиков прерываний!!! */
      xQueueSendToFrontFromISR(xPrintQueue, &(pcStringsToPrint[2]),
            &xHigherPriorityTaskWoken);
      iCount = 0;
   }
}
/*-----------------------------------------------------------*/

static void prvPrintTask(void *pvParameters) {
   int iIndexToString;

   /* Будет создано 2 экземпляра этой задачи. В качестве параметра
      при создании задачи выступает номер строки в таблице строк. */
   iIndexToString = (int) pvParameters;

   for (;;) {
      /* Вывести строку на консоль. Но не напрямую, а передав
         указатель на строку задаче-сторожу.*/
      xQueueSendToBack(xPrintQueue, &(pcStringsToPrint[iIndexT
oString]), 0);

      /* Блокировать задачу на промежуток времени случайной
         длины: от 0 до 500 квантов. */
      vTaskDelay((rand() % 500));
      /* Вообще функция rand() не является реентерабельной.
       Однако в этой программе это неважно. */
   }
}
```

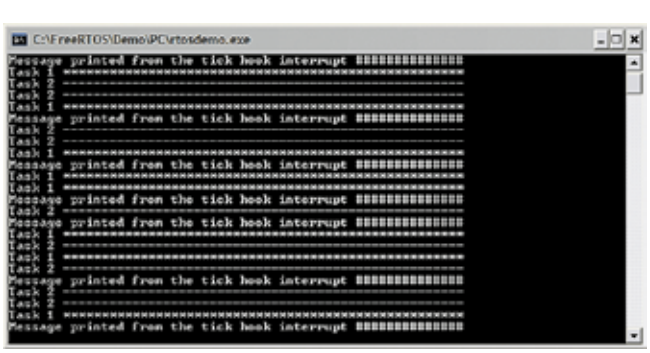

Рис. 9. Результат выполнения учебной программы № 3

Результат работы учебной программы № 3 приведен на рис. 9, на котором видно, что строки от двух задач с разными приоритетами и из тела обработчика прерывания выводятся на консоль без искажений. Следовательно, механизм взаимного исключения работает правильно.

Следует отметить, что в учебной программе № 3 приоритет задачи-сторожа задан самым низким в системе, поэтому строки накапливаются в очереди, пока задачи, их генерирующие, не заблокируются обе. В других ситуациях может потребоваться назначить задаче-сторожу более высокий приоритет. Это позволит ускорить прохождение очереди, но приведет к тому, что задача-сторож задержит выполнение более низкоприоритетных задач.

Выводы

В статье освещены вопросы организации совместного доступа к разделяемым ресурсам микроконтроллера. В дальнейших публикациях речь пойдет о сопрограммах — способе реализации многозадачной среды на микроконтроллерах с небольшим объемом оперативной памяти. Также внимание будет уделено нововведению версии FreeRTOS V7.0.0 — встроенной реализации программных таймеров.

Литература

1. Курниц А. FreeRTOS — операционная система для микроконтроллеров // Компоненты и технологии. 2011. № 2–7.
2. Barry R. Using the FreeRTOS real time kernel: A Practical Guide. 2009.
3. **http://www.freertos.org**
4. **http://ru.wikipedia.org/wiki/Реентерабельность**
5. **http://ru.wikipedia.org/wiki/Мьютекс**
6. **http://en.wikipedia.org/wiki/Reentrant_mutex**
7. **http://www.qnxclub.net/files/articles/invers/invers.pdf**
8. **http://ru.wikipedia.org/wiki/Взаимная_блокировка**
9. **http://www.ee.ic.ac.uk/t.clarke/rtos/lectures/RTOSlec2x2bw.pdf**