Продолжение. Начало в № 2`2011

FreeRTOS — операционная система для микроконтроллеров

Андрей КУРНИЦ
kurnits@stim.by

В этой статье мы продолжаем знакомить читателя с созданием программ, работающих под управлением FreeRTOS — операционной системы для микроконтроллеров.

Введение

Шестая часть статьи посвящена взаимодействию прерываний с остальной частью программы и поможет читателям ответить на следующие вопросы:

1) Какие API-функции и макросы можно использовать внутри обработчиков прерываний?
2) Как реализовать отложенную обработку прерываний?
3) Как создавать и использовать двоичные и счетные семафоры?
4) Как использовать очереди для передачи информации в обработчик прерывания и из него?
5) Каковы особенности обработки вложенных прерываний во FreeRTOS?

События и прерывания

Встраиваемые микроконтроллерные системы функционируют, отвечая действиями на события внешнего мира. Например, получение Ethernet-пакета (событие) требует обработки в задаче, которая реализует TCP/IP-стек (действие). Обычно встраиваемые системы обслуживают события, которые приходят от множества источников, причем каждое событие имеет свое требование по времени реакции системы и расходам времени на его обработку. При разработке встраиваемой микроконтроллерной системы необходимо подобрать свою стратегию реализации обслуживания событий внешнего мира. При этом перед разработчиком возникает ряд вопросов:

1) Каким образом события будут регистрироваться? Обычно применяют прерывания, однако возможен и опрос состояния выводов микроконтроллера.
2) В случае использования прерываний необходимо решить, какую часть программного кода, реализующего обработку события, поместить внутри обработчика прерывания, а какую — вне обработчика. Обычно стараются сократить размер обработчика прерывания настолько, насколько это возможно.
3) Как обработчики прерываний связаны с остальным кодом и как организовать программу, чтобы обеспечить наибыстрейшую обработку асинхронных событий внешнего мира?

FreeRTOS не предъявляет никаких требований к организации обработки событий, однако предоставляет удобные возможности для такой организации.

Прерывание (interrupt) — это событие (сигнал), заставляющее микроконтроллер изменить текущий порядок исполнения команд. При этом выполнение текущей последовательности команд приостанавливается, и управление передается обработчику прерывания — подпрограмме, которую можно представить функцией языка Си. Обработчик прерывания реагирует на событие и обслуживает его, после чего возвращает управление в прерванный код [6]. Прерывания инициируются периферией микроконтроллера, например прерывание от таймера/счетчика или изменение логического уровня на выводе микроконтроллера.

Следует заметить, что во FreeRTOS все API-функции и макросы, имена которых заканчиваются на FromISR или FROM_ISR, предназначены для использования в обработчиках прерываний и должны вызываться только внутри них.

Отложенная обработка прерываний

При проектировании встраиваемой микроконтроллерной системы на основе FreeRTOS необходимо учесть, насколько долго продолжается процесс обработки прерывания. В самом простом случае, когда при обработке прерывания повторные прерывания запрещены, временные задержки в обработчике прерываний могут существенно ухудшить время реакции системы на события. Тогда для выполнения продолжительных действий по обработке прерывания вводится так называемый «отложенный» режим их выполнения [5]. В процессе реакции на прерывание обработчик прерывания выполняет только первичные действия, например считывает данные. Затем львиную долю обработки берет на себя задача-обработчик прерывания. Такая организация обработки прерываний называется отложенной обработкой. При этом обработчик прерывания выполняет только самые «экстренные» действия, а основная обработка «откладывается», пока ее не выполнит задача-обработчик прерывания.

Двоичные семафоры

Двоичные семафоры предназначены для эффективной синхронизации выполнения задачи с возникновением прерывания. Они позволяют переводить задачу из состояния блокировки в состояние готовности к выполнению каждый раз, когда происходит прерывание. Это дает возможность перенести бóльшую часть кода, отвечающего за обработку внешнего события, из обработчика прерывания в тело задачи, выполнение которой синхронизировано с соответствующим прерыванием. Внутри обработчика прерывания останется лишь небольшой, быстро выполняющийся фрагмент кода. Говорят, что обработка прерывания отложена и непосредственно выполняется задачей-обработчиком.

Если прерывание происходит при возникновении особенно критичного к времени реакции внешнего события, то имеет смысл назначить задаче-обработчику достаточно высокий приоритет, чтобы при возникновении прерывания она вытесняла другие задачи в системе. Это произойдет, когда завершит свое выполнение обработчик прерывания. Выполнение задачи-обработчика начинается сразу же после окончания выполнения обработчика прерывания. Создается впечатление, что весь код, отвечающий за обработку внешнего события, реализован внутри обработчика прерывания (рис. 1).

На рис. 1 видно, что прерывание прерывает выполнение одной задачи и возвращает управление другой. В момент времени (1) выполняется прикладная задача, когда про-

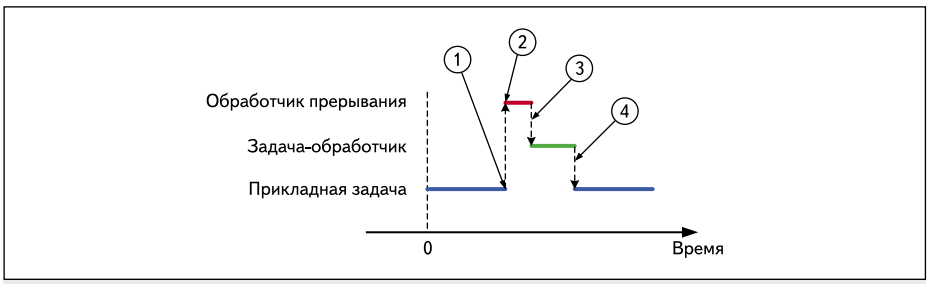

Рис. 1. Отложенная обработка прерывания с использованием двоичного семафора

исходит прерывание при возникновении какого-то внешнего события. В момент времени (2) управление получает обработчик прерывания, который, используя механизм двоичного семафора, выводит из блокированного состояния задачу-обработчик прерывания. Так как приоритет задачи-обработчика выше приоритета прикладной задачи, то задача-обработчик вытесняет прикладную задачу, которая остается в состоянии готовности к выполнению (3). В момент времени (4) задача-обработчик блокируется, ожидая возникновения следующего прерывания, и управление снова получает низкоприоритетная прикладная задача.

В теории многопоточного программирования [1] двоичный семафор определен как переменная, доступ к которой может быть осуществлен только с помощью двух атомарных функций (то есть тех, которые не могут быть прерваны планировщиком):

1) ***wait()*** или ***P()*** — означает захват семафора, если он свободен, и ожидание, если занят. В примере выше функцию ***wait()*** реализует задача-обработчик прерывания.
2) ***signal()*** или ***V()*** — означает выдачу семафора, то есть после того как одна задача выдает семафор, другая задача, которая ожидает возможности его захвата, может его захватить. В примере выше функцию ***signal()*** реализует обработчик прерывания.

Легко заметить, что операция выдачи двоичного семафора напоминает операцию помещения элемента в очередь, а операция захвата семафора — чтения элемента из очереди. Если установить размер очереди равным одному элементу, то очередь превращается в двоичный семафор. Наличие элемента в очереди означает, что одна (а может, и несколько) задача произвела(и) выдачу семафора, и теперь другая задача может его захватить. Пустая же очередь означает ситуацию, когда семафор уже был захвачен, и задача, которая «хочет» его захватить, вынуждена ожидать (находясь в блокированном состоянии), пока другая задача или обработчик прерывания произведут выдачу семафора.

В именах API-функций FreeRTOS для работы с семафорами используются термины ***Take*** — эквивалентен функции ***wait()***, то есть захват двоичного семафора, и ***Give*** — эквивалентен функции ***signal()***, то есть означает выдачу семафора.

На рис. 2 показано, как обработчик прерывания отдает семафор, вне зависимости от того, был ли он захвачен до этого. Задача-обработчик в свою очередь захватывает семафор, но никогда не отдает его обратно. Такой сценарий еще раз подчеркивает сходство работы двоичного семафора с очередью. Стоит отметить, что одна из частых причин ошибок в программе, связанных с семафорами, заключается в том, что в других сценариях задача после захвата семафора должна его отдать.

Работа с двоичными семафорами

Во FreeRTOS механизм семафоров основан на механизме очередей. По большому счету API-функции для работы с семафорами представляют собой макросы — «обертки» других API-функций для работы с очередями. Здесь и далее для простоты будем называть их API-функциями для работы с семафорами.

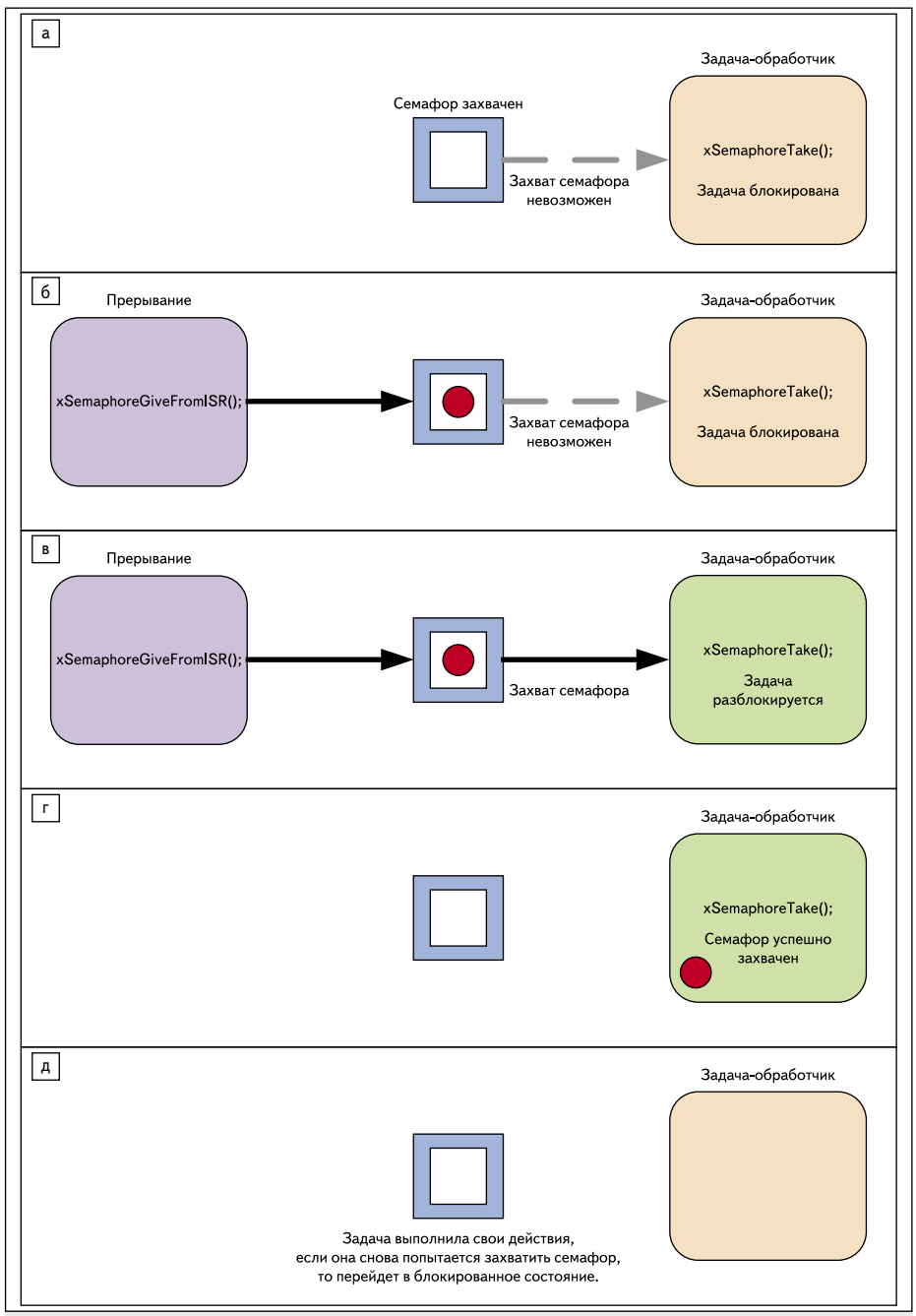

Рис. 2. Синхронизация прерывания и задачи-обработчика с помощью двоичного семафора

Все API-функции работы с семафорами сосредоточены в заголовочном файле ***/Source/Include/semphr.h***, поэтому следует убедиться, что этот файл находится в списке включенных (***#include***) в проект.

Доступ ко всем семафорам во FreeRTOS (а не только к двоичным) осуществляется с помощью дескриптора (идентификатора) — переменной типа ***xSemaphoreHandle***.

Создание двоичного семафора

Семафор должен быть явно создан перед первым его использованием. API-функция ***vSemaphoreCreateBinary()*** служит для создания двоичного семафора.

```
void vSemaphoreCreateBinary( xSemaphoreHandle xSemaphore );
```

Единственным аргументом является дескриптор семафора, в него будет возвращен дескриптор в случае успешного создания семафора. Если семафор не создан по причине отсутствия памяти, вернется значение NULL. Так как ***vSemaphoreCreateBinary()*** представляет собой макрос, то аргумент ***xSemaphore*** следует передавать напрямую, то есть нельзя использовать указатель на дескриптор и операцию переадресации.

Захват семафора

Осуществляется API-функцией ***xSemaphoreTake()*** и может вызываться только из задач. В классической терминологии [1] соответствует функции ***P()*** или ***wait()***. Чтобы задача смогла захватить семафор, он должен быть отдан другой задачей или обработчиком прерывания. Все типы семафоров за исключением рекурсивных (о них — в следующей публикации) могут быть захвачены с помощью ***xSemaphoreTake()***. API-функцию ***xSemaphoreTake()*** нельзя вызывать из обработчиков прерываний.

Прототип:

```
portBASE_TYPE xSemaphoreTake( xSemaphoreHandle xSemaphore,
portTickType xTicksToWait );
```

Назначение параметров и возвращаемое значение:

- ***xSemaphore*** — дескриптор семафора. Должен быть получен с помощью API-функции создания семафора.
- ***xTicksToWait*** — максимальное количество квантов времени, в течение которого задача может пребывать в блокированном состоянии, если семафор невозможно захватить (семафор недоступен). Для представления времени в миллисекундах следует использовать макроопределение portTICK_RATE_MS [2, КиТ № 4]). Задание ***xTicksToWait*** равным 0 приведет к тому, что задача не перейдет в блокированное состояние, если семафор недоступен, а продолжит свое выполнение сразу же. Установка ***xTicksToWait*** равным константе ***portMAX_DELAY*** приведет к тому, что выхода из блокированного состояния по истечении времени тайм-аута не произойдет. Задача будет сколь угодно долго «ожидать» возможности захватить семафор, пока такая возможность не появится. Для этого макроопределение ***INCLUDE_vTaskSuspend*** в файле ***FreeRTOSConfig.h*** должно быть равно «1».
- Возвращаемое значение — возможны два варианта:
  - ***pdPASS*** — свидетельствует об успешном захвате семафора. Если определено время тайм-аута (параметр ***xTicksToWait*** не равен 0), то возврат значения ***pdPASS*** говорит о том, что семафор стал доступен до истечения времени тайм-аута и был успешно захвачен.
  - ***pdFALSE*** — означает, что семафор недоступен (никто его не отдал). Если определено время тайм-аута (параметр ***xTicksToWait*** не равен 0 или ***portMAX_DELAY***), то возврат значения ***pdFALSE*** говорит о том, что время тайм-аута истекло, а семафор так и не стал доступен.

Выдача семафора из обработчика прерывания

Все типы семафоров во FreeRTOS, исключая рекурсивные, могут быть выданы из тела обработчика прерывания при помощи API-функции ***xSemaphoreGiveFromISR()***.

API-функция ***xSemaphoreGiveFromISR()*** представляет собой специальную версию API-функции ***xSemaphoreGive()***, которая предназначена для вызова из тела обработчика прерывания.

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

```
portBASE_TYPE xSemaphoreGiveFromISR( xSemaphoreHandle
xSemaphore, portBASE_TYPE *pxHigherPriorityTaskWoken );
```

Назначение параметров и возвращаемое значение:

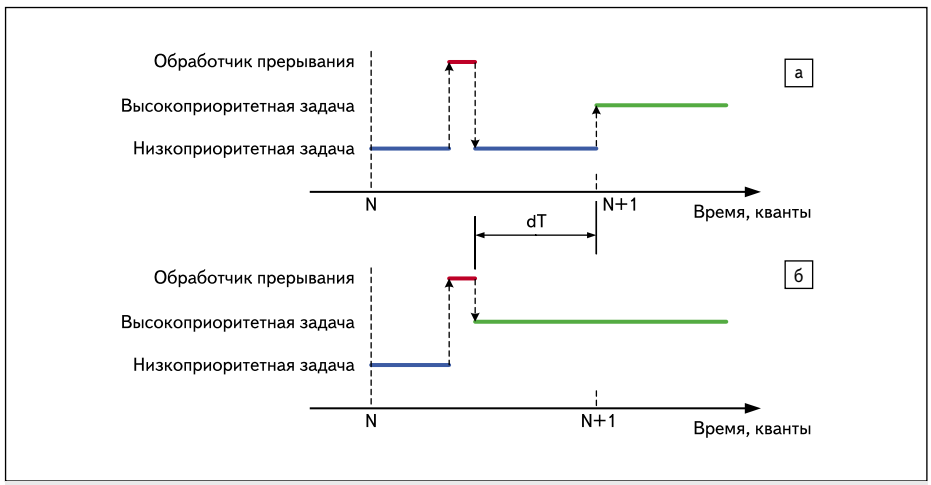

Рис. 3. Результат вызова xSemaphoreGiveFromISR():
а) без принудительного переключения контекста; б) с принудительным переключением контекста

1. ***xSemaphore*** — дескриптор семафора, который должен быть в явном виде создан до первого использования.
2. ***pxHigherPriorityTaskWoken*** — значение по адресу ***pxHigherPriorityTaskWoken*** устанавливает сама API-функция ***xSemaphoreGiveFromISR()*** в зависимости от того, разблокирована ли более высокоприоритетная задача в результате выдачи семафора. Подробнее об этом будет сказано далее.
3. Возвращаемое значение — возможны два варианта:
   - ***pdPASS*** — вызов ***xSemaphoreGiveFromISR()*** был успешным, семафор отдан.
   - ***pdFAIL*** — означает, что семафор в момент вызова ***xSemaphoreGiveFromISR()*** уже был доступен, то есть ранее отдан другой задачей или прерыванием.

Если после выдачи семафора в теле обработчика прерывания была разблокирована более высокоприоритетная задача, чем та, что была прервана обработчиком прерывания, то API-функция ***xSemaphoreGiveFromISR()*** установит ****pxHigherPriorityTaskWoken*** равным ***pdTRUE***. В противном случае значение ****pxHigherPriorityTaskWoken*** останется без изменений.

Значение ****pxHigherPriorityTaskWoken*** необходимо отслеживать для того, чтобы «вручную» выполнить переключение контекста задачи в конце обработчика прерывания, если в результате выдачи семафора была разблокирована более высокоприоритетная задача. Если этого не сделать, то после выполнения обработчика прерывания выполнение продолжит та задача, выполнение которой были прервано этим прерыванием (рис. 3). Ничего «страшного» в этом случае не произойдет: текущая задача будет выполняться до истечения текущего кванта времени, после чего планировщик выполнит переключение контекста (которое он выполняет каждый системный квант), и управление получит более высокоприоритетная задача (рис. 3а). Единственное, что пострадает,—

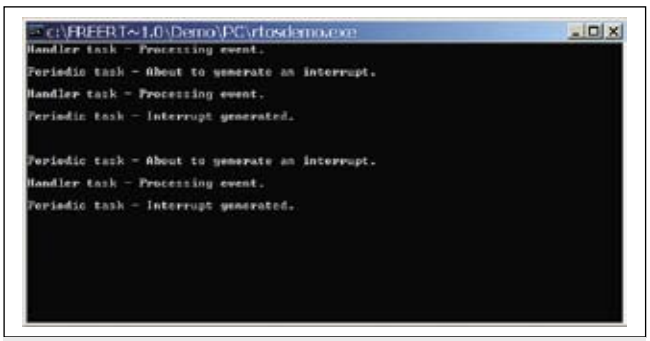

Рис. 4. Результаты выполнения учебной программы № 1

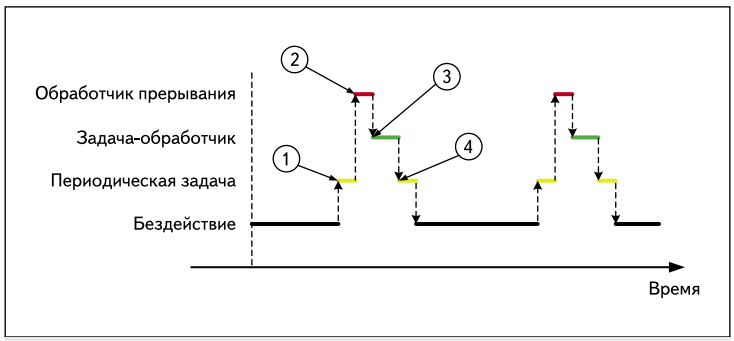

Рис. 5. Последовательность выполнения задач в учебной программе № 1

это время реакции системы на прерывание, которое может составлять до одного системного кванта: величина dT на рис. 3.

Далее в учебной программе № 1 будет приведен пример использования значения ***pxHigherPriorityTaskWoken*** для принудительного переключения контекста.

В случае использования API-функции ***xSemaphoreGive()*** переключение контекста происходит автоматически, и нет необходимости в его принудительном переключении.

Рассмотрим учебную программу № 1, в которой продемонстрировано использование двоичного семафора для синхронизации прерывания и задачи-обработчика этого прерывания:

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

/* Двоичный семафор – глобальная переменная */
xSemaphoreHandle xBinarySemaphore;

/*-----------------------------------------------------------*/
/* Периодическая задача */
static void vPeriodicTask(void *pvParameters) {
    for (;;) {
        * Эта задача используется только с целью генерации
          прерывания каждые 500 мс */
        vTaskDelay(500 / portTICK_RATE_MS);
        /* Сгенерировать прерывание.
          Вывести сообщение до этого и после. */
        puts("Periodic task - About to generate an interrupt.\r\n");
        __asm {int 0x82} /* Сгенерировать прерывание MS-DOS */
        puts("Periodic task - Interrupt generated.\r\n\r\n\r\n");
    }
}

/*-----------------------------------------------------------*/
/* Обработчик прерывания */
static void __interrupt __far vExampleInterruptHandler( void )
{
    static portBASE_TYPE xHigherPriorityTaskWoken;
    xHigherPriorityTaskWoken = pdFALSE;
    /* Отдать семафор задаче-обработчику */
    xSemaphoreGiveFromISR( xBinarySemaphore,
&xHigherPriorityTaskWoken );
    if( xHigherPriorityTaskWoken == pdTRUE )
    {
        /* Это разблокирует задачу-обработчик. При этом
           приоритет задачи-обработчика выше приоритета
           выполняющейся в данный момент периодической
           задачи. Поэтому переключаем контекст
           принудительно – так мы добьемся того, что после
           выполнения обработчика прерывания управление
           получит задача-обработчик.*/

        /* Макрос, выполняющий переключение контекста.
         * На других платформах имя макроса может быть другое! */
        portSWITCH_CONTEXT();
    }
}
```

```
/*-----------------------------------------------------------*/
/* Задача-обработчик */
static void vHandlerTask(void *pvParameters) {
    /* Как и большинство задач, реализована как бесконечный цикл */
    for (;;) {
        /* Реализовано ожидание события с помощью двоичного
           семафора. Семафор после создания становится
           доступен (так, как будто его кто-то отдал).
           Поэтому сразу после запуска планировщика задача
           захватит его. Второй раз сделать это ей не удастся,
           и она будет ожидать, находясь в блокированном
           состоянии, пока семафор не отдаст обработчик
           прерывания. Время ожидания задано равным
           бесконечности, поэтому нет необходимости проверять
           возвращаемое функцией xSemaphoreTake() значение. */
        xSemaphoreTake(xBinarySemaphore, portMAX_DELAY);
        /* Если программа "дошла" до этого места, значит,
           семафор был успешно захвачен.
           Обработка события, связанного с семафором.
           В нашем случае – индикация на дисплей. */
        puts("Handler task - Processing event.\r\n");
    }
}

/*-----------------------------------------------------------*/
/* Точка входа. С функции main() начнется выполнение
   программы. */
int main(void) {
    /* Перед использованием семафор необходимо создать. */
    vSemaphoreCreateBinary(xBinarySemaphore);
    /* Связать прерывание MS-DOS с обработчиком прерывания
vExampleInterruptHandler(). */
    _dos_setvect(0x82, vExampleInterruptHandler);
    /* Если семафор успешно создан */
    if (xBinarySemaphore != NULL) {
        /* Создать задачу-обработчик, которая будет
           синхронизирована с прерыванием.
           Приоритет задачи-обработчика выше,
           чем у периодической задачи. */
        xTaskCreate(vHandlerTask, "Handler", 1000, NULL, 3, NULL);
        /* Создать периодическую задачу, которая будет
           генерировать прерывание с некоторым интервалом.
           Ее приоритет – ниже, чем у задачи-обработчика. */
        xTaskCreate(vPeriodicTask, "Periodic", 1000, NULL, 1, NULL);
        /* Запуск планировщика. */
        vTaskStartScheduler();
    }
    /* При нормальном выполнении программа до этого места
       "не дойдет" */
    for (;;)
        ;
}
```

В демонстрационных целях использовано не аппаратное, а программное прерывание MS-DOS, которое «вручную» вызывается из служебной периодической задачи каждые 500 мс. Заметьте, что сообщение на дисплей выводится как до генерации прерывания, так и после него, что позволяет проследить последовательность выполнения задач (рис. 4).

Следует обратить внимание на использование параметра ***xHigherPriorityTaskWoken*** в API-функции ***xSemaphoreGiveFromISR()***. До вызова функции ему присваивается значение ***pdFALSE***, а после вызова — проверяется на равенство ***pdTRUE***. Таким образом отслеживается необходимость принудительного переключения контекста. В данной учебной программе такая необходимость возникает каждый раз, так как в системе постоянно находится более высокоприоритетная задача-обработчик, которая ожидает возможности захватить семафор.

Для принудительного переключения контекста служит API-макрос ***portSWITCH_CONTEXT()***. Однако для других платформ имя макроса будет иным, например, для микроконтроллеров AVR это будет ***taskYIELD()***, для ARM7 — ***portYIELD_FROM_ISR()***. Узнать точное имя макроса можно из демонстрационного проекта для конкретной платформы.

Переключение между задачами в учебной программе № 1 приведено на рис. 5.

Бóльшую часть времени ни одна задача не выполняется (бездействие), но каждые 0,5 с управление получает периодическая задача (1). Она выводит первое сообщение на экран и принудительно вызывает прерывание, об-

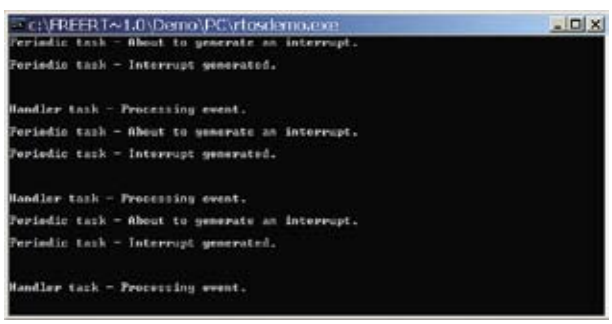

Рис. 6. Результаты выполнения учебной программы № 1 при отсутствии принудительного переключения контекста

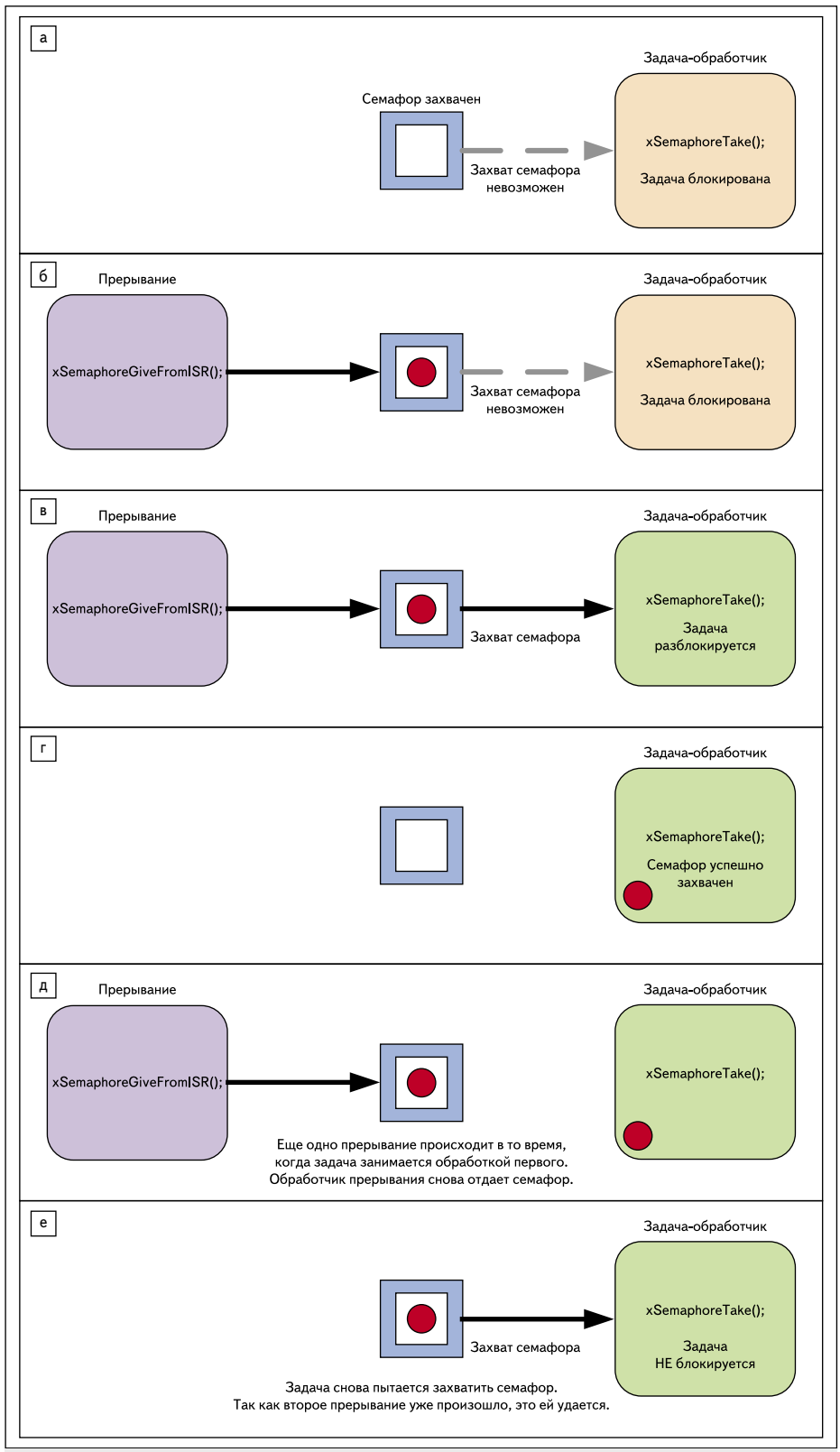

Рис. 7. «Потеря» прерывания при обработке с помощью двоичного семафора

работчик которого начинает выполняться сразу же (2). Обработчик прерывания отдает семафор, поэтому разблокируется задача-обработчик, которая ожидала возможности захватить этот семафор. Приоритет у задачи-обработчика выше, чем у периодической задачи, поэтому благодаря принудительному переключению контекста задача-обработчик получает управление (3). Задача-обработчик выводит свое сообщение на дисплей и пытается снова захватить семафор, который уже недоступен, поэтому она блокируется. Управление снова получает низкоприоритетная периодическая задача (4). Она выводит второе свое сообщение на дисплей и блокируется на время 0,5 с. Система снова переходит в состояние бездействия.

Если не выполнять принудительного переключения контекста, то есть исключить из программы строку:

```
portSWITCH_CONTEXT();
```

то можно наблюдать описанный ранее эффект (рис. 6).

В этом случае можно видеть, что сообщения, выводимые низкоприоритетной периодической задачей, следуют друг за другом, то есть высокоприоритетная задача-обработчик не получает управления сразу после того, как обработчик прерывания отдает семафор.

Подведя итоги, можно представить такую последовательность действий при отложенной обработке прерываний с помощью двоичного семафора:

- Происходит событие внешнего мира, вследствие него — прерывание микроконтроллера.
- Выполняется обработчик прерывания, который отдает семафор и разблокирует таким образом задачу — обработчик прерывания.
- Задача-обработчик начинает выполняться, как только завершит выполнение обработчик прерывания. Первое, что она делает, — захватывает семафор.
- Задача-обработчик обслуживает событие, связанное с прерыванием, после чего пытается снова захватить семафор и переходит в блокированное состояние, пока семафор снова не станет доступен.

Счетные семафоры

Организация обработки прерываний с помощью двоичных семафоров — отличное решение, если частота возникновения одного и того же прерывания не превышает некоторый порог. Если это же самое прерывание возникнет до того, как задача-обработчик завершит его обработку, то задача-обработчик не перейдет в блокированное состояние по завершении обработки предыдущего прерывания, а сразу же займется обслуживанием следующего. Предыдущее прерывание окажется потерянным. Этот сценарий показан на рис. 7.

Таким образом, с использованием двоичных семафоров из цепочки быстро следующих друг за другом событий может быть обслужено максимум одно событие.

Решить проблему обслуживания серии быстро следующих друг за другом событий можно используя счетные семафоры.

В отличие от двоичных семафоров состояние счетного семафора определяется не значениями отдан/захвачен, а представляет собой целое неотрицательное число — значение счетного семафора. И если двоич-

ный семафор — это, по сути, очередь длиной в 1 элемент, то счетный семафор можно представить очередью в несколько элементов. Причем текущее значение семафора представляет собой длину очереди, то есть количество элементов, которые в данный момент находятся в очереди. Значение элементов, хранящихся в очереди, когда она используется как счетный (или двоичный) семафор, не важно, а важно само наличие или отсутствие элемента.

Существует два основных применения счетных семафоров:

1. Подсчет событий. В этом случае обработчик прерывания будет отдавать семафор, то есть увеличивать его значение на единицу, когда происходит событие. Задача-обработчик будет захватывать семафор (уменьшать его значение на единицу) каждый раз при обработке события. Текущее значение семафора будет представлять собой разность между количеством событий, которые произошли, и количеством событий, которые обработаны. Такой способ организации взаимодействия показан на рис. 8. При создании счетного семафора для подсчета количества событий следует задавать начальное его значение, равное нулю.
2. Управление доступом к ресурсам. В этом случае значение счетного семафора представляет собой количество доступных ресурсов. Для получения доступа к ресурсу задача должна сначала получить (захватить) семафор — это уменьшит значение семафора на единицу. Когда значение семафора станет равным нулю, это означает, что доступных ресурсов нет. Когда задача завершает работу с данным ресурсом, она отдает семафор — увеличивает его значение на единицу. При создании счетного семафора для управления ресурсами следует задавать начальное его значение равным количеству свободных ресурсов. В дальнейших публикациях будет более подробно освещена тема управления ресурсами во FreeRTOS.

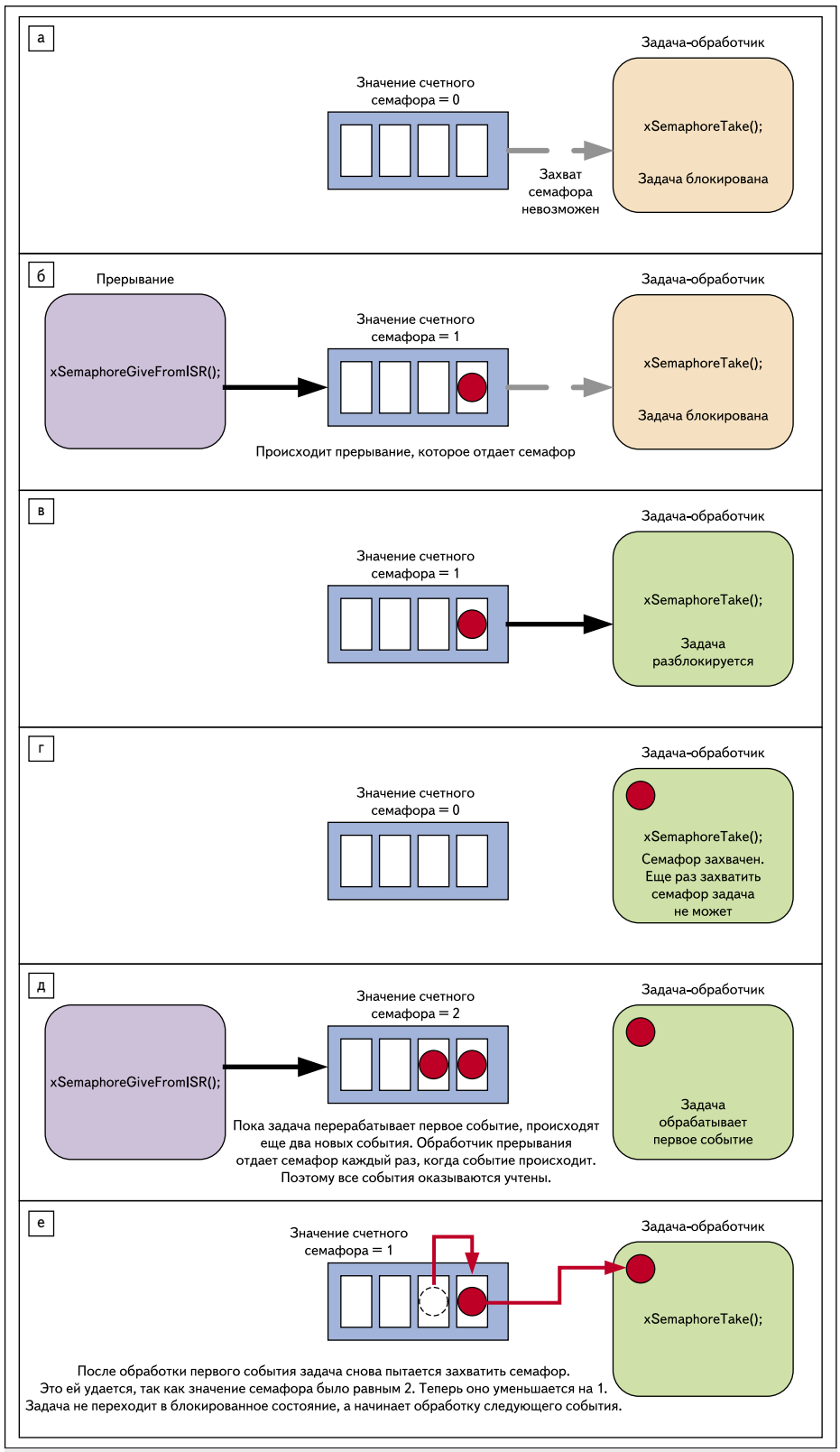

Рис. 8. Подсчет событий с помощью счетного семафора

Работа со счетными семафорами

Создание счетного семафора

Как и другие объекты ядра, счетный семафор должен быть явно создан перед первым его использованием:

```
xSemaphoreHandle xSemaphoreCreateCounting( unsigned portBASE_
TYPE uxMaxCount,
unsigned portBASE_TYPE uxInitialCount );
```

Назначение параметров и возвращаемое значение:

1. ***uxMaxCount*** — задает максимально возможное значение семафора. Если проводить аналогию с очередями, то он эквивалентен размеру очереди. Определяет максимальное количество событий, которые может обработать семафор, или общее количество доступных ресурсов, если семафор используется для управления ресурсами.
2. ***uxInitialCount*** — задает значение семафора, которое он принимает сразу после создания. Если семафор используется для подсчета событий, следует установить ***uxInitialCount*** равным 0, что будет означать, что ни одного события еще не произошло. Если семафор используется для управления доступом к ресурсам, то следует установить ***uxInitialCount*** равным максимальному значению — параметру ***uxMaxCount***. Это будет означать, что все ресурсы свободны.

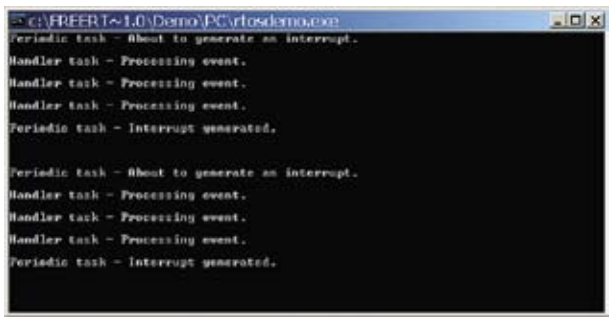

Рис. 9. Обработка быстро следующих событий

3. Возвращаемое значение — равно NULL, если семафор не создан по причине отсутствия требуемого объема свободной памяти. Ненулевое значение означает успешное создание счетного семафора. Это значение необходимо сохранить в переменной типа ***xSemaphoreHandle*** для обращения к семафору в дальнейшем.

API-функции выдачи (инкремента, увеличения на единицу) и захвата (декремента, уменьшения на единицу) счетного семафора ничем не отличаются от таковых для двоичных семафоров: ***xSemaphoreTake()*** — захват семафора; ***xSemaphoreGive()***, ***xSemaphoreGiveFromISR()*** — выдача семафора, соответственно, из задачи и из обработчика прерывания.

Продемонстрировать работу со счетными семафорами можно слегка модифицировав учебную программу № 1, приведенную выше. Изменению подвергнется функция, реализующая прерывание:

```
/*-----------------------------------------------------------*/
/* Обработчик прерывания */
static void __interrupt __far vExampleInterruptHandler( void )
{
    static portBASE_TYPE xHigherPriorityTaskWoken;
    xHigherPriorityTaskWoken = pdFALSE;
    /* Отдать семафор задаче-обработчику несколько раз.
       Таким образом симулируется быстро следующая группа
       событий, с которыми связано прерывание. Первая выдача
       разблокирует задачу-обработчик. Последующие будут
       "запомнены" счетным семафором и обработаны позже.
       "Потери" событий не происходит. */
    xSemaphoreGiveFromISR( xBinarySemaphore,
&xHigherPriorityTaskWoken );
    xSemaphoreGiveFromISR( xBinarySemaphore,
&xHigherPriorityTaskWoken );
    xSemaphoreGiveFromISR( xBinarySemaphore,
&xHigherPriorityTaskWoken );
    if( xHigherPriorityTaskWoken == pdTRUE )
    {

        /* Макрос, выполняющий переключение контекста.
        * На других платформах имя макроса может быть другое! */
        portSWITCH_CONTEXT();
    }
}
```

API-функцию создания двоичного семафора в главной функции ***main()***:

```
/* Перед использованием семафор необходимо создать. */
vSemaphoreCreateBinary(xBinarySemaphore);
```

следует заменить функцией создания счетного семафора:

```
/* Перед использованием счетный семафор необходимо создать.
   Семафор сможет обработать максимум 10 событий. Начальное
значение = 0. */
   xBinarySemaphore = xSemaphoreCreateCounting( 10, 0 );
```

В модифицированном варианте искусственно создаются три быстро следующих друг за другом события. Каждому событию соответствует операция выдачи (инкремента) счетного семафора. Задача-обработчик, как и ранее, обрабатывает события, выполняя операцию захвата (декремента) семафора. Результат выполнения модифицированной учебной программы № 1 приведен на рис. 9.

Судя по результатам работы (рис. 9), все три события были обработаны задачей-обработчиком. Если же изменить тип используемого в программе семафора на двоичный, то результат выполнения программы не будет отличаться от приведенного на рис. 4. Это будет свидетельствовать о том, что двоичный семафор в отличие от счетного не может зафиксировать более одного события.

Использование очередей в обработчиках прерываний

Как было показано выше, семафоры предназначены для передачи факта наступления события между задачами и прерываниями. Очереди же можно использовать как для передачи событий, так и для передачи данных.

Ранее [2, КиТ № 6] мы говорили об API-функциях для работы с очередями: ***xQueueSendToFront()***, ***xQueueSendToBack()*** и ***xQueueReceive()***. Использование их внутри тела обработчика прерывания приведет к краху программы. Для этого существуют версии этих функций, предназначенные для вызова из обработчиков прерываний: ***xQueueSendToFrontFromISR()***, ***xQueueSendToBackFromISR()*** и ***xQueueReceiveFromISR()***, причем вызов их из тела задачи запрещен. API-функция ***xQueueSendFromISR()*** является полным эквивалентом функции ***xQueueSendToBackFromISR()***.

Функции ***xQueueSendToFrontFromISR()***, ***xQueueSendToBackFromISR()*** служат для записи данных в очередь и отличаются лишь тем, что первая помещает элемент в начало очереди, а вторая — в конец. В остальном их поведение идентично.

Рассмотрим их прототипы:

```
portBASE_TYPE xQueueSendToFrontFromISR( xQueueHandle
xQueue, void *pvItemToQueue portBASE_TYPE
*pxHigherPriorityTaskWoken );

portBASE_TYPE xQueueSendToBackFromISR( xQueueHandle
xQueue, void *pvItemToQueue portBASE_TYPE
*pxHigherPriorityTaskWoken );
```

Аргументы и возвращаемое значение:

1. ***xQueue*** — дескриптор очереди, в которую будет записан элемент. Дескриптор очереди может быть получен при ее создании API-функцией ***xQueueCreate()***.
2. ***pvItemToQueue*** — указатель на элемент, который будет записан в очередь. Размер элемента зафиксирован при создании очереди, так что для побайтового копирования элемента достаточно иметь указатель на него.
3. ***pxHigherPriorityTaskWoken*** — значение ****pxHigherPriorityTaskWoken*** устанавливается равным ***pdTRUE***, если существует задача, которая «хочет» прочитать данные из очереди, и приоритет у нее выше, чем у задачи, выполнение которой прервало прерывание. Если таковой задачи нет, то значение ****pxHigherPriorityTaskWoken*** остается неизменным. Проанализировав значение ****pxHigherPriorityTaskWoken*** после выполнения ***xQueueSendToFrontFromISR()*** или ***xQueueSendToBackFromISR()***, можно сделать вывод о необходимости принудительного переключения контекста в конце обработчика прерывания. В этом случае управление сразу перейдет разблокированной высокоприоритетной задаче.
4. Возвращаемое значение — может принимать 2 значения:
   - ***pdPASS*** — означает, что данные успешно записаны в очередь.
   - ***errQUEUE_FULL*** — означает, что данные не записаны в очередь, так как очередь заполнена.

API-функция ***xQueueReceiveFromISR()*** служит для чтения данных с начала очереди. Вызываться она должна только из обработчиков прерываний.

Ее прототип:

```
portBASE_TYPE xQueueReceiveFromISR(
        xQueueHandle pxQueue,
        void *pvBuffer,
        portBASE_TYPE *pxTaskWoken
);
```

Аргументы и возвращаемое значение:

1. ***xQueue*** — дескриптор очереди, из которой будет считан элемент. Дескриптор очереди может быть получен при ее создании API-функцией ***xQueueCreate()***.
2. ***pvBuffer*** — указатель на область памяти, в которую будет скопирован элемент из очереди. Объем памяти, на которую ссылается указатель, должен быть не меньше размера одного элемента очереди.

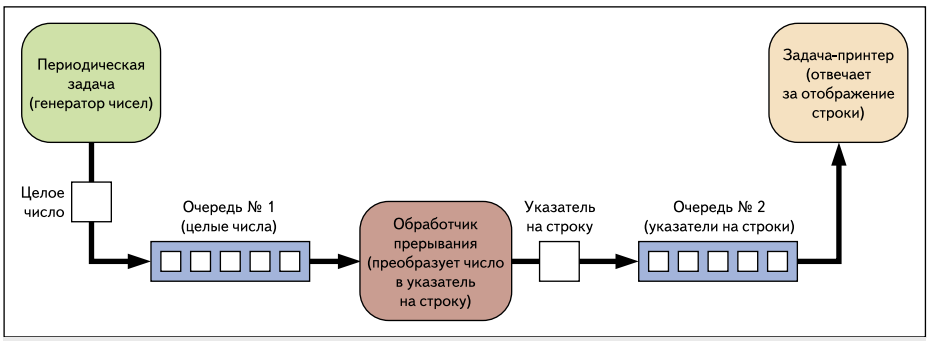

Рис. 10. Обмен данными между задачами и прерыванием в учебной программе № 2

3. ***pxTaskWoken*** — значение ****pxTaskWoken*** устанавливается равным ***pdTRUE***, если существует задача, которая «хочет» записать данные в очередь, и приоритет у нее выше, чем у задачи, выполнение которой прервало прерывание. Если таковой задачи нет, то значение ****pxTaskWoken*** остается неизменным. Проанализировав значение ****pxTaskWoken*** после выполнения ***xQueueReceiveFromISR()***, можно сделать вывод о необходимости принудительного переключения контекста в конце обработчика прерывания. В этом случае управление сразу перейдет разблокированной высокоприоритетной задаче.
4. Возвращаемое значение — может принимать 2 значения:
   – ***pdTRUE*** — означает, что данные успешно прочитаны из очереди.
   – ***pdFALSE*** — означает, что данные не прочитаны, так как очередь пуста.

Следует обратить внимание, что в отличие от версий API-функций для работы с очередями, предназначенными для вызова из тела задачи, описанные выше API-функции не имеют параметра ***portTickType xTicksToWait***, который задает время ожидания задачи в блокированном состоянии. Что и понятно, так как обработчик прерывания — это не задача, и он не может переходить в блокированное состояние. Поэтому если чтение/запись из/в очередь невозможно выполнить внутри обработчика прерывания, то соответствующая API-функция вернет управление сразу же.

Эффективное использование очередей

Бóльшая часть демонстрационных проектов из дистрибутива FreeRTOS содержит пример работы с очередями, в котором очередь используется для передачи каждого отдельного символа, полученного от универсального асинхронного приемопередатчика (UART), где символ записывается в очередь внутри обработчика прерывания, а считывается из нее в теле задачи.

Передача сообщения побайтно при помощи очереди — это очень неэффективный метод обмена информацией (особенно на высоких скоростях передачи) и приводится в демонстрационных проектах лишь для наглядности.

Гораздо эффективнее использовать один из следующих подходов:

1. Внутри обработчика прерывания помещать каждый принятый символ в простой буфер, а когда сообщение будет принято полностью или обнаружится окончание передачи, использовать двоичный семафор для разблокировки задачи-обработчика, которая произведет интерпретацию принятого сообщения.
2. Интерпретировать сообщение внутри обработчика прерывания, а очередь использовать для передачи интерпретированной команды (как показано на рис. 5, КиТ № 6`2011, стр. 102). Такой подход допускается, если интерпретация не содержит сложных алгоритмов и занимает немного процессорного времени.

Рассмотрим учебную программу № 2, в которой продемонстрировано применение API-функций ***xQueueSendToBackFromISR()*** и ***xQueueReceiveFromISR()*** внутри обработчика прерываний. В программе реализована задача — генератор чисел, которая отвечает за генерацию последовательности целых чисел. Целые числа по 5 штук помещаются в очередь № 1, после чего происходит программное прерывание (для простоты оно генерируется из тела задачи — генератора чисел). Внутри обработчика прерывания происходит чтение числа из очереди № 1 с помощью API-функции ***xQueueReceiveFromISR()***. Далее это число преобразуется в указатель на строку, который помещается в очередь № 2 с помощью API-функции ***xQueueSendToBackFromISR()***. Задача-принтер считывает указатели из очереди № 2 и выводит соответствующие им строки на экран (рис. 10).

Текст учебной программы № 2:

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

/* Дескрипторы очередей – глобальные переменные */
xQueueHandle xIntegerQueue;
xQueueHandle xStringQueue;

/*-----------------------------------------------------------*/
/* Периодическая задача — генератор чисел */
static void vIntegerGenerator(void *pvParameters) {
   portTickType xLastExecutionTime;
   unsigned portLONG ulValueToSend = 0;
   int i;
   /* Переменная xLastExecutionTime 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(). */
   xLastExecutionTime = xTaskGetTickCount();
   for (;;) {
      /* Это периодическая задача. Период выполнения – 200 мс. */
      vTaskDelayUntil(&xLastExecutionTime, 200 / portTICK_RATE_MS);
      /* Отправить в очередь № 1 5 чисел от 0 до 4. Числа будут
         считаны из очереди в обработчике прерывания.
         Обработчик прерывания всегда опустошает очередь, поэтому
         запись 5 элементов будет всегда возможна – в переходе
         в блокированное состояние нет необходимости */
      for (i = 0; i < 5; i++) {
         xQueueSendToBack(xIntegerQueue, &ulValueToSend, 0);
         ulValueToSend++;
      }
      /* Принудительно вызвать прерывание. Отобразить
         сообщение до его вызова и после. */
      puts("Generator task - About to generate an interrupt.");
      __asm {int 0x82} /* Эта инструкция сгенерирует прерывание. */
      puts("Generator task - Interrupt generated.\r\n");
   }
}

/*-----------------------------------------------------------*/
/* Обработчик прерывания */
static void __interrupt __far vExampleInterruptHandler( void )
{
   static portBASE_TYPE xHigherPriorityTaskWoken;
   static unsigned long ulReceivedNumber;
   /* Массив строк определен как static, значит, память для его
      размещения выделяется как
      для глобальной переменной (он хранится не в стеке). */
   static const char *pcStrings[] =
   {
      "String 0",
      "String 1",
      "String 2",
      "String 3"
   };
   /* Аргумент API-функции xQueueReceiveFromISR(), который
      устанавливается в pdTRUE, если операция с очередью
      разблокирует более высокоприоритетную задачу.
      Перед вызовом xQueueReceiveFromISR() должен
      принудительно устанавливаться в pdFALSE */
   xHigherPriorityTaskWoken = pdFALSE;
   /* Считывать из очереди числа, пока та не станет пустой. */
   while( xQueueReceiveFromISR( xIntegerQueue,
            &ulReceivedNumber,
            &xHigherPriorityTaskWoken ) != errQUEUE_EMPTY )
   {
      /* Обнулить в числе все биты, кроме последних двух.
         Таким образом, полученное число будет принимать
         значения от 0 до 3. Использовать полученное число
         как индекс в массиве строк. Получить таким образом
         указатель на строку, который передать в очередь № 2 */
      ulReceivedNumber &= 0x03;
      xQueueSendToBackFromISR( xStringQueue,
         &pcStrings[ ulReceivedNumber ],
         &xHigherPriorityTaskWoken );
   }
   /* Проверить, не разблокировалась ли более высокоприоритетная
      задача при записи в очередь. Если да, то выполнить
      принудительное переключение контекста. */

   if( xHigherPriorityTaskWoken == pdTRUE )
   {
      /* Макрос, выполняющий переключение контекста.
       На других платформах имя макроса может быть другое! */
      portSWITCH_CONTEXT();
   }
}

/*-----------------------------------------------------------*/
/* Задача-принтер. */
static void vStringPrinter(void *pvParameters) {
   char *pcString;
   /* Бесконечный цикл */
   for (;;) {
      /* Прочитать очередной указатель на строку из очереди № 2.
         Находится в блокированном состоянии сколь угодно долго,
         пока очередь № 2 пуста. */
      xQueueReceive(xStringQueue, &pcString, portMAX_DELAY);
      /* Вывести строку, на которую ссылается указатель на дисплей. */
      puts(pcString);
   }
}

/*-----------------------------------------------------------*/
/* Точка входа. С функции main() начнется выполнение
   программы. */
int main(void) {
   /* Как и другие объекты ядра, очереди необходимо создать
      до первого их использования. Очередь xIntegerQueue будет
      хранить переменные типа unsigned long. Очередь
```

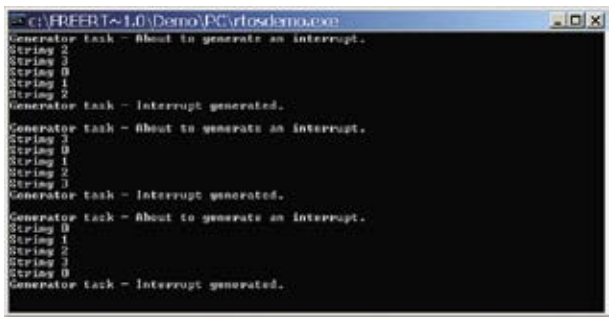

Рис. 11. Результаты выполнения учебной программы № 2

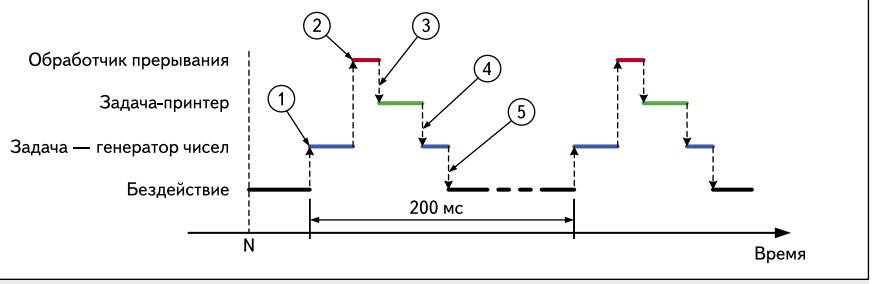

Рис. 12. Последовательность выполнения задач и прерываний в учебной программе № 2

```
        xStringQueue будет хранить переменные типа char* –
        указатели на нуль-терминальные строки.
        Обе очереди создаются размером 10 элементов.
        Реальная программа должна проверять значения xIntegerQueue,
        xStringQueue, чтобы убедиться, что очереди успешно созданы. */
    xIntegerQueue = xQueueCreate(10, sizeof(unsigned long));
    xStringQueue = xQueueCreate(10, sizeof(char *));
    /* Связать прерывание MS-DOS с обработчиком прерывания
vExampleInterruptHandler(). */
    _dos_setvect(0x82, vExampleInterruptHandler);
    /* Создать задачу — генератор чисел с приоритетом 1. */
    xTaskCreate(vIntegerGenerator, "IntGen", 1000, NULL, 1, NULL);
    /* Создать задачу-принтер с приоритетом 2. */
    xTaskCreate(vStringPrinter, "String", 1000, NULL, 2, NULL);
    /* Запуск планировщика. */
    vTaskStartScheduler();
    /* При нормальном выполнении программа до этого места
       "не дойдет" */
    for (;;)
        ;
}
```

Заметьте, что для эффективного распределения ресурсов памяти данных (как и рекомендовалось в [2, КиТ № 6]) очередь № 2 хранит не сами строки, а лишь указатели на строки, которые содержатся в отдельном массиве. Такое решение вполне допустимо, так как содержимое строк в программе не изменяется.

По результатам выполнения (рис. 11) видно, что в результате возникновения прерывания была разблокирована высокоприоритетная задача-принтер, после чего управление снова возвращается низкоприоритетной задаче — генератору чисел (рис. 12).

Задача-бездействие выполняется бóльшую часть времени. Каждые 200 мс она вытесняется задачей — генератором чисел (1). Задача — генератор чисел записывает в очередь № 1 пять целых чисел, после чего принудительно вызывает прерывание (2). Обработчик прерывания считывает числа из очереди № 1 и записывает в очередь № 2 указатели на соответствующие строки. Запись в очередь № 2 разблокирует высокоприоритетную задачу-принтер (3). Задача-принтер считывает указатели на строки из очереди № 2, пока они там есть, и выводит соответствующие строки на экран. Как только очередь № 2 опустошится, задача-принтер переходит в блокированное состояние (4). Управление снова получает низкоприоритетная задача — генератор чисел, которая также блокируется на время ~200 мс, так что система снова переходит в состояние бездействия (5).

Вложенность прерываний

Во многих архитектурах микроконтроллеров прерывания имеют приоритеты, которые могут быть жестко заданы, но может существовать возможность и конфигурировать уровни приоритетов прерываний.

Важно различать приоритет задач и приоритет прерываний. Приоритеты прерываний аппаратно фиксированы в архитектуре микроконтроллера (или определены при его конфигурации), а приоритеты задач — это программная абстракция на уровне ядра FreeRTOS. Приоритет прерываний задает преимущество на выполнение того или иного обработчика прерывания при возникновении сразу нескольких прерываний. Задачи не выполняются во время выполнения обработчика прерывания, поэтому приоритет задач не имеет никакого отношения к приоритету прерываний.

Под вложенностью прерываний понимается корректная работа FreeRTOS при одновременном возникновении сразу нескольких прерываний с разными приоритетами, когда обработчик низкоприоритетного прерывания еще не завершился, а возникает высокоприоритетное прерывание, и процессор начинает выполнять его программу-обработчик.

Большинство портов FreeRTOS допускает вложение прерываний. Эти порты требуют задания одного или двух конфигурационных макроопределений в файле ***FreeRTOSConfig.h***:

1. ***configKERNEL_INTERRUPT_PRIORITY*** — задает приоритет прерывания, используемого для отсчета системных квантов FreeRTOS. Если порт не использует макроопределение ***configMAX_SYSCALL_INTERRUPT_PRIORITY***, то для обеспечения вложенности прерываний все прерывания, в обработчиках которых встречаются API-функции FreeRTOS, должны иметь этот же приоритет.
2. ***configMAX_SYSCALL_INTERRUPT_PRIORITY*** — задает наибольший приоритет прерывания, из обработчика которого можно вызывать API-функции FreeRTOS (чтобы прерывания могли быть вложенными).

Получить модель вложенности прерываний без каких-либо ограничений можно задав значение ***configMAX_SYSCALL_INTERRUPT_PRIORITY*** выше, чем ***configKERNEL_INTERRUPT_PRIORITY***.

Рассмотрим пример. Пусть некий микроконтроллер имеет 7 возможных приоритетов прерываний. Значение приоритета 7 соответствует самому высокоприоритетному прерыванию, 1 — самому низкоприоритетному. Зададим значение ***configMAX_SYSCALL_INTERRUPT_PRIORITY*** = 3, а значение ***configKERNEL_INTERRUPT_PRIORITY*** = 1 (рис. 13).

Прерывания с приоритетом 1–3 не будут выполняться, пока ядро или задача выполняют код, находящийся в критической секции, но могут при этом использовать API-функции. На время реакции на такие прерывания будет оказывать влияние активность ядра FreeRTOS.

На прерывания с приоритетом 4 и выше не влияют критические секции, так что ничего, что делает ядро в данный момент, не помешает выполнению обработчика такого прерывания. Обычно те прерывания, которые имеют самые строгие временны́е требования (например, управление током в обмотках двигателя), должны иметь приоритет выше, чем ***configMAX_SYSCALL_INTERRUPT_PRIORITY***, чтобы гарантировать, что ядро не внесет дрожание (jitter) во время реакции на прерывание.

И наконец, прерывания, которые не вызывают никаких API-функций, могут иметь любой из возможных приоритетов.

Критическая секция в FreeRTOS — это участок кода, во время выполнения которого запрещены прерывания процессора и, соответственно, не происходит переключение контекста каждый квант времени [7]. Подробнее о критических секциях — в следующей публикации.

Следует отметить, что в популярном семействе микроконтроллеров ARM Cortex M3 (как и в некоторых других) меньшие значения приоритетов прерываний соответствуют логически бóльшим приоритетам. Если вы хотите назначить прерыванию более высокий приоритет, вы назначаете ему приоритет с более низким номером. Одна из возможных причин краха программы в таких случаях — назначение прерыванию номера приоритета меньшего, чем ***configMAX_SYSCALL_INTERRUPT_PRIORITY***, и вызов из него API-функции.

Пример корректной настройки файла ***FreeRTOSConfig.h*** для микроконтроллеров ARM Cortex M3:

```
#define configKERNEL_INTERRUPT_PRIORITY          255
#define configMAX_SYSCALL_INTERRUPT_PRIORITY     191
```

Выводы

В любой операционной системе реального времени с вытесняющей многозадачностью существует потенциальный источник ошибок и сбоев работы системы — это единовременное обращение сразу нескольких задач к одному ресурсу. В качестве ресурса может выступать множество видов объектов:

- память;
- периферийные устройства;
- библиотечные функции и др.

Проблема возникает, когда одна задача начинает какие-либо действия с ресурсом, но не успевает их закончить, когда происходит переключение контекста и управление получает другая задача, которая обращается к тому же самому ресурсу, состояние которого носит промежуточный, не окончательный характер (из-за воздействия первой задачи).

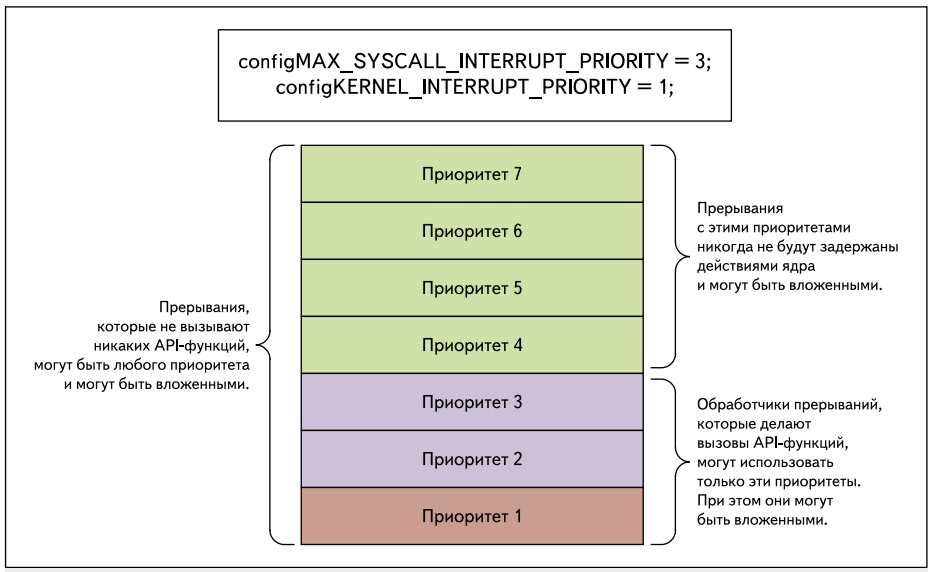

Рис. 13. Возможность вызова API-функций в обработчиках прерываний

При этом результат обращения к ресурсу в обеих задачах окажется ошибочным, искаженным.

К счастью, во FreeRTOS существуют встроенные на уровне ядра механизмы обеспечения совместного доступа к одному аппаратному ресурсу. С применением счетных семафоров для управления доступом к ресурсам читатель уже познакомился. В следующей публикации внимание будет сконцентрировано на средствах FreeRTOS обеспечения безопасного доступа к ресурсам. К таковым относятся:

- мьютексы и двоичные семафоры;
- счетные семафоры;
- критические секции;
- задачи-сторожа (gatekeeper tasks).

Литература

1. Эндрюс Г. Р. Основы многопоточного, параллельного и распределенного программирования. Пер. с англ. М.: ИД «Вильямс», 2003.
2. Курниц А. FreeRTOS — операционная система для микроконтроллеров // Компоненты и технологии. 2011. № 2–6.
3. Barry R. Using the FreeRTOS real time kernel: A Practical Guide. 2009.
4. http://www.freertos.org
5. http://www.ignatova-e-n.narod.ru/mop/zag6.html
6. http://ru.wikipedia.org/wiki/Прерывание
7. http://www.mikrocontroller.net/attachment/95930/FreeRTOSPaper.pdf