Продолжение. Начало в № 2`2011

FreeRTOS — операционная система для микроконтроллеров

Мы продолжаем изучение FreeRTOS — операционной системы для микроконтроллеров. В пятой части статьи основное внимание сфокусировано на очередях — безопасном механизме взаимодействия задач друг с другом. Будут показаны опасности организации взаимодействия между задачами «напрямую» и обосновано применение очередей, а также рассказано об основных принципах, заложенных в функционирование очередей. Читатель узнает о том, как создать очередь, как записать данные в очередь и прочитать их оттуда. Будут освещены вопросы целесообразного выбора типа данных, хранящихся в очереди, и назначения приоритетов задачам, которые записывают и считывают данные из очереди.

Андрей КУРНИЦ
kurnits@stim.by

Необходимость использования очередей

Самый простой способ организовать обмен информацией между задачами — использовать общую глобальную переменную. Доступ к такой переменной осуществляется одновременно из нескольких задач. Такой подход был продемонстрирован в [1, КиТ № 4] в учебной программе № 3.

Однако такой подход имеет существенный недостаток: при совместном доступе нескольких задач к общей переменной возникает ситуация, когда выполнение одной задачи прерывается планировщиком именно в момент модификации общей переменной, когда та содержит не окончательное (искаженное) значение. При этом результат работы другой задачи, которая получит управление и обратится к этой переменной, также окажется искаженным.

Продемонстрировать этот эффект позволяет учебная программа № 1, в которой объявлена глобальная переменная ***lVal*** и две задачи: задача, которая модифицирует общую переменную, — ***vModifyTask()***, и задача, которая проверяет значение этой переменной, — ***vCheckTask()***. Модификация производится так, чтобы итоговое значение глобальной переменной после окончания вычислений не изменялось. В случае если значение общей переменной отличается от первоначального, задача ***vCheckTask()*** выдает соответствующее сообщение на экран.

Текст учебной программы № 1:

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

/* Глобальная переменная, доступ к которой будет
 * осуществляться из нескольких задач */
long lVal = 100;

/*-----------------------------------------------------------*/
/* Функция, реализующая задачи, которая модифицирует
 * глобальную переменную */
void vModifyTask(void *pvParameters) {

    /* Бесконечный цикл */
    for (;;) {
        /* Модифицировать переменную lVal так,
         * чтобы ее значение не изменилось */
        lVal += 10;
        lVal -= 10;
    }
}

/*-----------------------------------------------------------*/
/* Функция, реализующая задачу, которая проверяет значение
 * переменной */
void vCheckTask(void *pvParameters) {

    /* Бесконечный цикл */
    for (;;) {
        if (lVal != 100) {
            puts("Variable lVal is not 100!");
        }
        vTaskDelay(100);
    }
}

/*-----------------------------------------------------------*/
/* Точка входа. С функции main() начнется выполнение
 * программы. */
int main(void) {
    /* Создать задачи с равным приоритетом */
    xTaskCreate(vModifyTask, "Modify", 1000, NULL, 1, NULL);
    xTaskCreate(vCheckTask, "Check", 1000, NULL, 1, NULL);
    /* Запуск планировщика. Задачи начнут выполняться. */
    vTaskStartScheduler();
    for (;;);
}
```

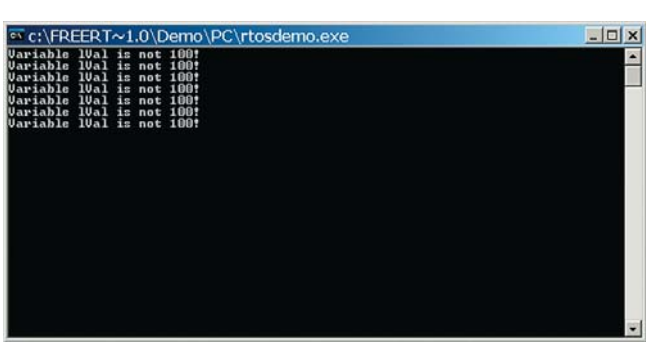

Рис. 1. Результат выполнения учебной программы № 1

Результаты работы показывают (рис. 1), что значение глобальной переменной часто оказывается не равным ожидаемому (100).

Решить подобные проблемы позволяет использование очередей для передачи информации между задачами. Во FreeRTOS очереди представляют собой фундаментальный механизм взаимодействия задач друг с другом. Они могут быть использованы для передачи информации как между задачами, так и между прерываниями и задачами. Основное преимущество использования очередей — это то, что их использование является безо-

пасным в многозадачной среде (thread safe). То есть при использовании очередей автоматически решается проблема совместного доступа нескольких задач к одному аппаратному ресурсу, роль которого в данном случае играет память.

Характеристики очередей

Хранение информации в очереди

Информация хранится в очереди в виде элементов (items) — блоков памяти фиксированного размера. В качестве элемента очереди может выступать любая переменная языка Си. В случае если это переменная типа char, размер блока будет равен 1 байту, если это структура или массив, размер блока будет равен, соответственно, размеру структуры или массива.

Элементы очереди в контексте обмена информацией между задачами будем называть сообщениями.

Запись элемента в очередь приводит к созданию побайтовой копии элемента в очереди. Побайтовое копирование происходит и при чтении элемента из очереди.

Очередь может хранить в себе конечное число элементов фиксированного размера. Максимальное число элементов, которое может хранить очередь, называется размером очереди. Как размер элемента, так и размер очереди задаются при создании очереди и остаются неизменными до ее удаления.

Важно отметить, что память выделяется сразу под все элементы очереди, то есть пустая и заполненная очередь не отличаются друг от друга по объему занимаемой памяти. При записи элементов в очередь динамического выделения памяти не происходит.

Очередь функционирует по принципу «первым вошел — первым вышел» (First In First Out, FIFO), то есть элемент, который раньше остальных был помещен в очередь (в конец очереди), будет и прочитан раньше остальных (рис. 2). Обычно элементы записываются в конец («хвост») очереди и считываются с начала («головы») очереди.

На рис. 2а показаны очередь длиной 5 элементов для хранения целочисленных переменных, Задача 1, которая будет записывать элементы в очередь, и Задача 2, которая будет считывать элементы из очереди. В исходном состоянии очередь не содержит ни одного элемента, то есть пуста.

На рис. 2б Задача 1 записывает число «15» в конец очереди. Так как теперь очередь содержит 1 элемент, то он является одновременно и началом, и концом очереди.

На рис. 2в Задача 1 записывает еще один элемент («69») в конец очереди. Теперь очередь содержит 2 элемента, причем элемент «15» находится в начале очереди, а элемент «69» — в конце.

На рис. 2г Задача 2 считывает элемент, находящийся в начале очереди, то есть элемент «15». Таким образом, выполняется принцип

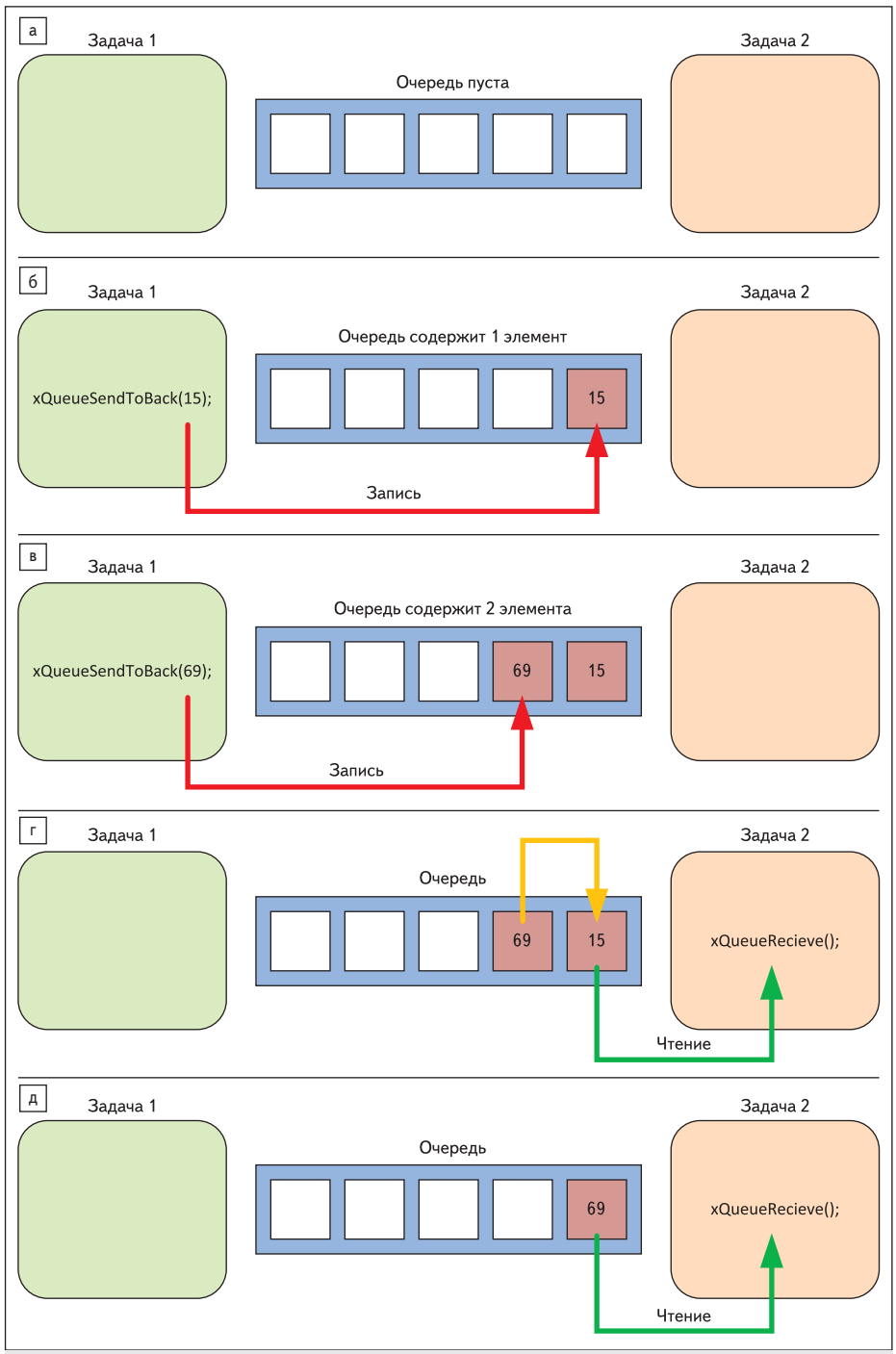

Рис. 2. Запись и чтение элементов из очереди по принципу FIFO

«первым вошел — первым вышел», так как элемент «15» первым записан в очередь и, соответственно, первым из нее считан. Теперь очередь снова содержит один элемент («69») в начале очереди, который и будет считан при следующем чтении из очереди Задачей 2 (рис. 2д).

Следует отметить, что на рис. 2 показано использование API-функций для работы с очередями в упрощенном виде. Корректное их применение будет описано ниже.

Также имеется возможность помещать элементы в начало очереди, тогда очередь превращается в стек, работающий по принципу «последним вошел — первым вышел» (Last In First Out, LIFO).

Доступ из множества задач

Очередь — это самостоятельный объект ядра, она не принадлежит ни одной конкретной задаче. Напротив, любое количество задач могут как читать, так и записывать данные в одну и ту же очередь. Следует отметить, что ситуация, когда в очередь помещают данные сразу несколько задач, является «обычным делом» для программ под управлением ОСРВ, однако чтение данных несколькими задачами из одной очереди встречается редко.

Блокировка при чтении из очереди

Когда задача пытается прочитать данные из очереди, которая не содержит ни одного элемента, то задача переходит в блокированное состояние. Такая задача вернется в состояние готовности к выполнению, если другая задача (или прерывание) поместит данные в очередь.

Выход из блокированного состояния возможен также при истечении тайм-аута, если очередь на протяжении этого времени оставалась пуста.

Данные из очереди могут читать сразу несколько задач. Когда очередь пуста, то все они находятся в блокированном состоянии. Когда в очереди появляется элемент данных, начнет выполняться та задача, которая имеет наибольший приоритет.

Возможна ситуация, когда равноприоритетные задачи ожидают появления данных в очереди. В этом случае при поступлении данных в очередь управление получит та задача, время ожидания которой было наибольшим. Остальные же останутся в блокированном состоянии.

Блокировка при записи в очередь

Как и при чтении из очереди, задача может находиться в блокированном состоянии, ожидая возможность записи в очередь. Это происходит, когда очередь полностью заполнена и в ней нет свободного места для записи нового элемента данных. До тех пор пока какая-либо другая задача не прочитает данные из очереди, задача, которая пишет в очередь, будет «ожидать», находясь в блокированном состоянии.

В одну очередь могут писать сразу несколько задач, поэтому возможна ситуация, когда несколько задач находятся в блокированном состоянии, ожидая завершения операции записи в одну очередь. Когда в очереди появится свободное место, получит управление задача с наивысшим приоритетом. В случае если запись в очередь ожидали равноприоритетные задачи, управление получит та, которая находилась в блокированном состоянии дольше остальных.

Работа с очередями

Теперь целесообразно рассмотреть конкретные API-функции FreeRTOS для работы с очередями. Все API-функции для работы с очередями используют переменную типа ***xQueueHandle***, которая служит в качестве дескриптора (идентификатора) конкретной очереди из множества всех очередей в программе. Дескриптор очереди можно получить при ее создании.

Создание очереди

Очередь должна быть явно создана перед первым ее использованием. API-функция ***xQueueCreate()*** служит для создания очереди, она возвращает переменную типа ***xQueueHandle*** в случае успешного создания очереди:

```
xQueueHandle xQueueCreate(unsigned portBASE_TYPE uxQueueLength,
                          unsigned portBASE_TYPE uxItemSize );
```

Ее параметры и возвращаемое значение:

- ***uxQueueLength*** — определяет размер очереди, то есть максимальное количество элементов, которые может хранить очередь.
- ***uxItemSize*** — задает размер одного элемента очереди в байтах, его легко получить с помощью оператора ***sizeof()***.
- Возвращаемое значение — дескриптор очереди. Равен ***NULL***, если очередь не создана по причине отсутствия достаточного объема памяти в куче FreeRTOS. Ненулевое значение свидетельствует об успешном создании очереди, в этом случае оно должно быть сохранено в переменной типа ***xQueueHandle*** для дальнейшего обращения к очереди.

При создании очереди ядро FreeRTOS выделяет блок памяти из кучи для ее размещения. Этот блок памяти используется как для хранения элементов очереди, так и для хранения служебной структуры управления очередью, которая представлена структурой ***xQUEUE***.

Получить точный размер структуры ***xQUEUE*** для конкретной платформы и компилятора можно, получив значение следующего выражения:

```
sizeof(xQUEUE)
```

При этом следует учесть, что структура ***xQUEUE*** используется ядром в собственных целях и доступа к этой структуре из текста прикладных исходных файлов (***main.c*** в том числе) изначально нет. Чтобы получить доступ к структуре ***xQUEUE***, необходимо включить в исходный файл строку:

```
#include "..\\queue.c"
```

Для платформы x86 и компилятора Open Watcom (которые используются в учебных программах) размер структуры ***xQUEUE*** составляет 58 байт.

Запись элемента в очередь

Для записи элемента в конец очереди используется API-функция ***xQueueSendToBack()***, для записи элемента в начало очереди — ***xQueueSendToFront()***. Так как запись в конец очереди применяется гораздо чаще, чем в начало, то вызов API-функции ***xQueueSend()*** эквивалентен вызову ***xQueueSendToBack()***. Прототипы у всех трех API-функций одинаковы:

```
portBASE_TYPE xQueueSendXXXX (xQueueHandle xQueue,
                              const void * pvItemToQueue,
                              portTickType xTicksToWait );
```

Назначение параметров и возвращаемое значение:

- ***xQueue*** — дескриптор очереди, в которую будет записан элемент. Дескриптор очереди может быть получен при ее создании API-функцией ***xQueueCreate()***.
- ***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.
- ***xTicksToWait*** — максимальное количество квантов времени, в течение которого задача может пребывать в блокированном состоянии, если очередь полна и записать новый элемент невозможно. Для представления времени в миллисекундах следует использовать макроопределение ***portTICK_RATE_MS*** [1, КиТ № 4]. Задание ***xTicksToWait*** равным «0» приведет к тому, что задача не перейдет в блокированное состояние, если очередь полна, а продолжит свое выполнение. Установка ***xTicksToWait*** равным константе ***portMAX_DELAY*** приведет к тому, что выхода из блокированного состояния по истечении тайм-аута не будет. Задача будет сколь угодно долго «ожидать» возможности записать элемент в очередь, пока такая возможность не появится. При этом макроопределение ***INCLUDE_vTaskSuspend*** в файле ***FreeRTOSConfig.h*** должно быть равно «1».
- Возвращаемое значение — может возвращать 2 значения:
  - ***pdPASS*** — означает, что данные успешно записаны в очередь. Если определена продолжительность тайм-аута (параметр ***xTicksToWait*** не равен «0»), то возврат значения ***pdPASS*** говорит о том, что свободное место в очереди появилось до истечения тайм-аута и элемент был помещен в очередь.
  - ***errQUEUE_FULL*** — означает, что данные не записаны в очередь, так как очередь заполнена. Если определена продолжительность тайм-аута (параметр ***xTicksToWait*** не равен «0» или ***portMAX_DELAY***), то возврат значения ***errQUEUE_FULL*** говорит о том, что тайм-аут завершен и свободное место в очереди так и не появилось.

Следует отметить, что API-функции ***xQueueSendToBack()*** и ***xQueueSendToFront()*** нельзя вызывать из тела обработчика прерывания. Для этой цели служат специальные версии этих API-функций — ***xQueueSendToBackFromISR()***и ***xQueueSendToFrontFromISR()*** соответственно. Более подробно об использовании API-функций FreeRTOS в теле обработчика прерывания будет рассказано в дальнейших публикациях.

Чтение элемента из очереди

Чтение элемента из очереди может быть произведено двумя способами:

- Элемент считывается из очереди (создается его побайтовая копия в другую переменную), после чего он удаляется из очереди. Именно такой способ считывания продемонстрирован на рис. 2.
- Создается побайтовая копия элемента, при этом элемент из очереди не удаляется.

Для считывания элемента с удалением его из очереди используется API-функция ***xQueueReceive()***. Ее прототип:

```
portBASE_TYPE xQueueReceive(
                    xQueueHandle xQueue,
                    const void * pvBuffer,
                    portTickType xTicksToWait);
```

Для считывания элемента из очереди без его удаления используется API-функция ***xQueuePeek()***. Ее прототип:

```
portBASE_TYPE xQueuePeek(
                    xQueueHandle xQueue,
                    const void * pvBuffer,
                    portTickType xTicksToWait);
```

Назначение параметров и возвращаемое значение для API-функций ***xQueueReceive()*** и ***xQueuePeek()*** одинаковы:

- ***xQueue*** — дескриптор очереди, из которой будет прочитан элемент. Дескриптор очереди может быть получен при ее создании API-функцией ***xQueueCreate()***.
- ***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.
- ***xTicksToWait*** — максимальное количество квантов времени, в течение которого задача может пребывать в блокированном состоянии, если очередь не содержит ни одного элемента. Для представления времени в миллисекундах следует использовать макроопределение ***portTICK_RATE_MS*** [1, КиТ № 4]. Задание ***xTicksToWait*** равным «0» приведет к тому, что задача не перейдет в блокированное состояние, а продолжит свое выполнение, если очередь в данный момент пуста. Установка ***xTicksToWait*** равным константе ***portMAX_DELAY*** приведет к тому, что выхода из блокированного состояния по истечении тайм-аута не будет. Задача будет сколь угодно долго «ожидать» появления элемента в очереди. При этом макроопределение ***INCLUDE_vTaskSuspend*** в файле ***FreeRTOSConfig.h*** должно быть равно «1».
- Возвращаемое значение — может возвращать 2 значения:
  - ***pdPASS*** — означает, что данные успешно прочитаны из очереди. Если определена продолжительность тайм-аута (параметр ***xTicksToWait*** не равен «0»), то возврат значения ***pdPASS*** говорит о том, что элемент в очереди появился (или уже был там) до истечения тайм-аута и был успешно прочитан.
  - ***errQUEUE_EMPTY*** — означает, что элемент не прочитан из очереди, так как очередь пуста. Если определена продолжительность тайм-аута (параметр ***xTicksToWait*** не равен «0» или ***portMAX_DELAY***), то возврат значения ***errQUEUE_FULL*** говорит о том, что тайм-аут завершен и никакая другая задача или прерывание не записали элемент в очередь.

Как и в случае с записью элемента в очередь, API-функции ***xQueueReceive()*** и ***xQueuePeek()*** нельзя вызывать из тела обработчика прерывания. Для этих целей служит API-функция ***xQueueReceiveFromISR()***, которая будет описана в следующих публикациях.

Состояние очереди

Получить текущее количество записанных элементов в очереди можно с помощью API-функции ***uxQueueMessagesWaiting()***:

```
unsigned portBASE_TYPE uxQueueMessagesWaiting(xQueueHandle xQueue);
```

Назначение параметров и возвращаемое значение:

- ***xQueue*** — дескриптор очереди, состояние которой необходимо получить. Дескриптор очереди может быть получен при ее создании API-функцией ***xQueueCreate()***.
- Возвращаемое значение — количество элементов, которые хранит очередь в момент вызова ***uxQueueMessagesWaiting()***. Если очередь пуста, то возвращаемым значением будет «0».

Как и в случаях с чтением и записью элемента в очередь, API-функцию ***uxQueueMessagesWaiting()*** нельзя вызывать из тела обработчика прерывания. Для этих целей служит API-функция ***uxQueueMessagesWaitingFromISR()***.

Удаление очереди

Если в программе использована схема распределения памяти, допускающая удаление задач [1, КиТ № 5], то полезной окажется возможность удалить и очередь, которая использовалась для взаимодействия с удаленной задачей. Для удаления очереди служит API-функция ***vQueueDelete()***. Ее прототип:

```
void vQueueDelete(xQueueHandle xQueue);
```

Единственный аргумент — это дескриптор удаляемой очереди. При успешном завершении API-функция ***vQueueDelete()*** освободит всю память, выделенную как для размещения служебной структуры управления очередью, так и для размещения самих элементов очереди.

Рассмотреть процесс обмена сообщениями между несколькими задачами можно на примере учебной программы № 2, в которой реализована очередь для хранения элементов типа ***long***. Данные в очередь записывают две задачи-передатчика, а считывает данные одна задача-приемник.

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

/* Объявить переменную-дескриптор очереди.
 * Эта переменная будет использоваться
 * для работы с очередью из тела всех трех задач. */
xQueueHandle xQueue;

/*-----------------------------------------------------------*/
/* Функция, реализующая задачи-передатчики */
void vSenderTask(void *pvParameters) {
   /* Переменная, которая будет хранить значение, передаваемое
    * в очередь */
   long lValueToSend;
   /* Переменная, которая будет хранить результат выполнения
    * xQueueSendToBack() */
   portBASE_TYPE xStatus;
   /* Будет создано несколько экземпляров задачи. В качестве
    * параметра задачи выступает число, которое задача будет
    * записывать в очередь */
   lValueToSend = (long) pvParameters;
   /* Бесконечный цикл */
   for (;;) {
      /* Записать число в конец очереди.
       * 1-й параметр — дескриптор очереди, в которую будет
       * производиться запись, очередь создана до запуска
       * планировщика, и ее дескриптор сохранен в глобальной \
       * переменной xQueue.
       * 2-й параметр — указатель на переменную, которая будет
       * записана в очередь, в данном случае — lValueToSend.
       * 3-й параметр — продолжительность тайм-аута.
       * В данном случае задана равной 0, что соответствует
       * отсутствию времени ожидания, если очередь полна.
       * Однако из-за того, что задача-приемник сообщений имеет
       * более высокий приоритет, чем задачи-передатчики,
       * в очереди не может находиться более одного элемента.
       * Таким образом, запись нового элемента будет всегда
       * возможна. */
      xStatus = xQueueSendToBack(xQueue, &lValueToSend, 0);
      if (xStatus != pdPASS) {
         /* Если попытка записи не была успешной —
          * индицировать ошибку. */
         puts("Could not send to the queue.\r\n");
      }
      /* Сделать принудительный вызов планировщика, позволив,
       * таким образом, выполняться другой задаче-передатчику.
       * Переключение на другую задачу произойдет быстрее,
       * чем окончится текущий квант времени. */
      taskYIELD();
   }
}

/*-----------------------------------------------------------*/
/* Функция, реализующая задачу-приемник */
void vReceiverTask(void *pvParameters) {
   /* Переменная, которая будет хранить значение, полученное
    * из очереди */
   long lReceivedValue;
   /* Переменная, которая будет хранить результат выполнения
    * xQueueReceive() */
   portBASE_TYPE xStatus;
   /* Бесконечный цикл */
   for (;;) {
      /* Индицировать состояние, когда очередь пуста */
      if (uxQueueMessagesWaiting(xQueue) != 0) {
         puts("Queue should have been empty!\r\n");
      }
      /* Прочитать число из начала очереди.
       * 1-й параметр — дескриптор очереди, из которой будет
       * происходить чтение, очередь создана до запуска
       * планировщика, и ее дескриптор сохранен в глобальной
       * переменной xQueue.
       * 2-й параметр — указатель на буфер, в который будет
       * помещено число из очереди.
       * В данном случае — указатель на переменную lReceivedValue.
       * 3-й параметр — продолжительность тайм-аута, в течение
       * которого задача будет находиться в блокированном
       * состоянии, пока очередь пуста. В данном случае
       * макроопределение portTICK_RATE_MS используется
       * для преобразования времени 100 мс в количество
       * системных квантов.
       */
      xStatus = xQueueReceive(xQueue, &lReceivedValue, 100 /
portTICK_RATE_MS);
      if (xStatus == pdPASS) {
         /* Число успешно принято, вывести его на экран */
         printf("Received = %ld\r\n", lReceivedValue);
```

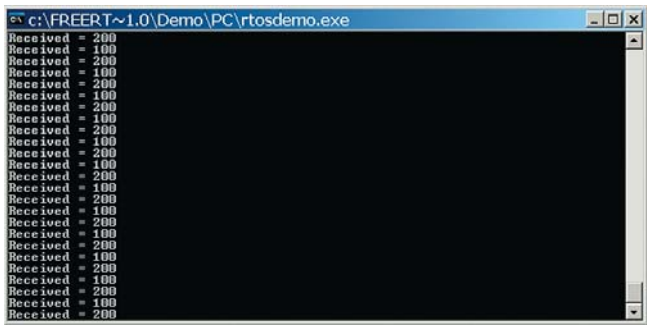

Рис. 3. Результат выполнения учебной программы № 2

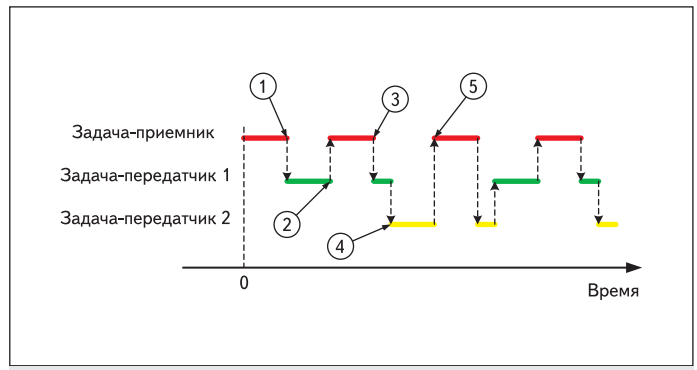

Рис. 4. Последовательность выполнения задач в учебной программе № 2

```
    } else {
        /* Данные не были прочитаны из очереди на протяжении
        * тайм-аута 100 мс.
        * При условии наличия нескольких задач-передатчиков
        * означает аварийную ситуацию
        */
        puts("Could not receive from the queue.\r\n");
    }
  }
}


/*-----------------------------------------------------------*/
/* Точка входа. С функции main() начнется выполнение
 * программы. */
int main(void) {
  /* Создать очередь размером 5 элементов для хранения
  * переменных типа long.
  * Размер элемента установлен равным размеру переменной
  * типа long.
  * Дескриптор созданной очереди сохранить в глобальной
  * переменной xQueue.
  */
  xQueue = xQueueCreate(5, sizeof(long));
  /* Если очередь успешно создана (дескриптор не равен NULL) */
  if (xQueue != NULL) {
    /* Создать 2 экземпляра задачи-передатчика. Параметр,
    * передаваемый задаче при ее создании, используется для
    * передачи экземпляру конкретного значения, которое
    * экземпляр задачи будет записывать в очередь.
    * Задача-передатчик 1 будет записывать значение 100.
    * Задача-передатчик 2 будет записывать значение 200.
    * Обе задачи создаются с приоритетом 1.
    */
    xTaskCreate(vSenderTask, "Sender1", 1000, (void *) 100, 1, NULL);
    xTaskCreate(vSenderTask, "Sender2", 1000, (void *) 200, 1, NULL);
    /* Создать задачу-приемник, которая будет считывать числа
     * из очереди.
     * Приоритет = 2, т.е. выше, чем у задач-передатчиков.
     */
    xTaskCreate(vReceiverTask, "Receiver", 1000, NULL, 2, NULL);
    /* Запуск планировщика. Задачи начнут выполняться. */
    vTaskStartScheduler();
  } else {
    /* Если очередь не создана */
  }
  /* При успешном создании очереди и запуске планировщика
   * программа никогда "не дойдет" до этого места. */
  for (;;)
    ;
}
```

Результат выполнения учебной программы № 2 показан на рис. 3, на котором видно, что задача-приемник получает сообщения от обеих задач-передатчиков.

В момент времени (0) (рис. 4) происходит запуск планировщика, который переводит в состояние выполнения задачу с наивысшим приоритетом — задачу-приемник. В момент времени (1) задача-приемник пытается прочитать элемент из очереди, однако очередь после создания пуста, и задача-приемник переходит в блокированное состояние до момента появления данных в очереди либо до момента истечения тайм-аута 100 мс. В состояние выполнения переходит одна из задач-передатчиков, какая именно — точно сказать нельзя, так как они имеют одинаковый приоритет, в нашем случае пусть это будет задача-передатчик 1. В момент времени (2) задача-передатчик 1 записывает число 100 в очередь. В этот момент выходит из блокированного состояния задача-приемник, так как она «ожидала» появления данных в очереди и приоритет ее выше. Прочитав данные из очереди, она вновь блокируется, так как очередь снова пуста (момент времени (3)). Управление возвращается прерванной задаче-передатчику 1, которая выполняет API-функцию вызова планировщика ***taskYIELD()***, в результате чего управление получает равноприоритетная задача-передатчик 2 (момент времени (4)). Когда она записывает значение 200 в очередь, снова разблокируется высокоприоритетная задача-приемник — момент времени (5), и цикл повторяется.

Следует отметить, что в ранее приведенном примере, когда задача-приемник имеет более высокий приоритет, чем задачи-передатчики, очередь не может быть заполнена более чем на 1 элемент данных.

Использование очередей для передачи составных типов

Одним из типичных способов организации обмена между задачами с применением очередей является организация нескольких задач-источников сообщений и одной задачи-приемника сообщений (как и в учебной программе выше). При этом полезной окажется возможность знать, какая именно задача-источник поместила данные в очередь, чтобы понять, какие именно действия нужно выполнить с этими данными.

Простой способ достижения этого — использовать в качестве элемента очереди структуру, в которой хранятся как сами данные, так и указан источник сообщения. На рис. 5 показана организация обмена информацией между задачами в абстрактной программе, реализующей контроллер двигателя с CAN-интерфейсом.

На рис. 5 изображена также структура ***xData***, которая выступает в данном случае типом элементов очереди. Структура содержит два целочисленных значения:

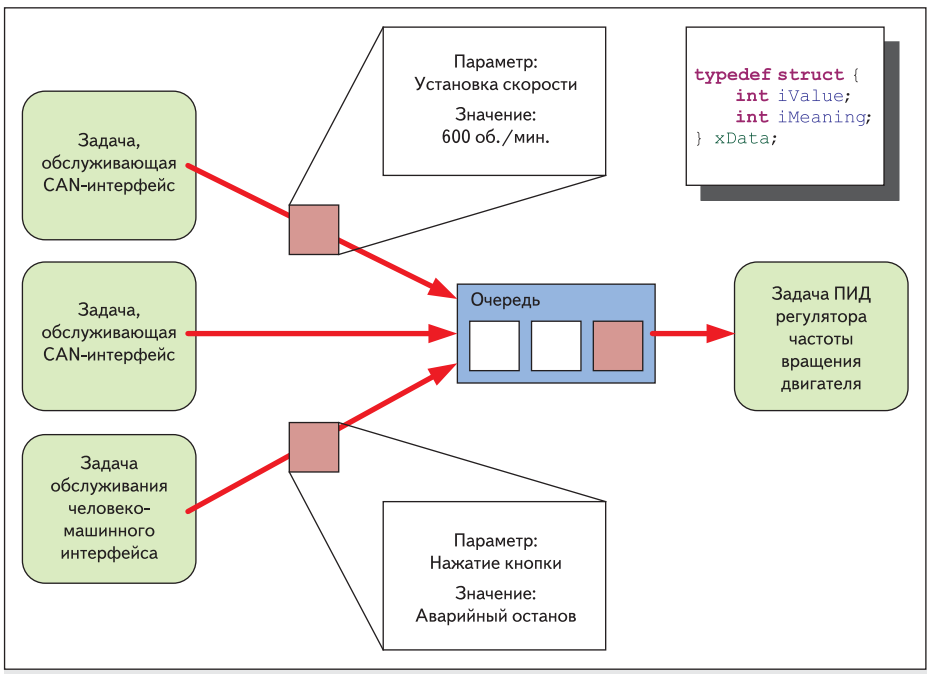

Рис. 5. Пример организации обмена информацией между задачами

- ***iMeaning*** — значение, смысл передаваемого через очередь параметра;
- ***iValue*** — числовое значение параметра.

Задача ПИД-регулятора частоты вращения двигателя ответственна за главную функцию устройства — поддержание частоты вращения на заданном уровне. Задача ПИД-регулятора должна соответствующим образом реагировать на действия оператора и команды по CAN-интерфейсу, она получает информацию о внешних воздействиях, считывая сообщения из очереди.

Задача обслуживания CAN-интерфейса отвечает за обработку входящих по CAN-шине сообщений, декодирует их и посылает сообщение в виде структуры ***xData*** в задачу ПИД-регулятора. Значение члена структуры ***iMeaning*** «установка скорости» позволяет задаче ПИД-регулятора определить, что значение ***iValue***, равное 600, есть не что иное, как новое значение уставки скорости вращения.

Задача обслуживания человеко-машинного интерфейса ответственна за взаимодействие оператора с контроллером двигателя. Оператор может вводить значения параметров, давать команды контроллеру, наблюдать его текущее состояние. Когда оператор нажал кнопку аварийной остановки двигателя, задача обслуживания человеко-машинного интерфейса сформировала соответствующую структуру ***xData***. Поле ***iMeaning*** указывает на нажатие оператором некоторой кнопки, а поле ***iValue*** — уточняет какой именно: кнопки аварийного останова. Такого рода сообщения (связанные с возникновением аварийной ситуации) целесообразно помещать не в конец, а в начало очереди, так, чтобы задача ПИД-контроллера обработала их раньше остальных находящихся в очереди, сократив, таким образом, время реакции системы.

Рассмотрим учебную программу № 3, в которой, как и в учебной программе № 2, будет две задачи-передатчика сообщений и одна задача-приемник. Однако в качестве единицы передаваемой информации на этот раз выступает структура, которая содержит сведения о задаче, которая передала это сообщение. Кроме того, продемонстрирована другая схема назначения приоритетов задачам, когда задача-приемник имеет более низкий приоритет, чем задачи-передатчики.

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

/* Номера функций-передатчиков сообщений */
#define mainSENDER_1    1
#define mainSENDER_2    2

/* Объявить переменную-дескриптор очереди. Эта переменная
 * будет использоваться для ссылки на очередь после ее создания. */
xQueueHandle xQueue;

/* Определить структуру, которая будет элементом очереди */
typedef struct
{
    unsigned char ucValue;
    unsigned char ucSource;
} xData;

/* Определить массив из двух структур, которые будут
 * записываться в очередь */
static const xData xStructsToSend[ 2 ] =
{
    { 100, mainSENDER_1 }, /* Используется задачей-передатчиком 1 */
    { 200, mainSENDER_2 } /* Используется задачей-передатчиком 2 */
};

/*-----------------------------------------------------------*/
/* Функция, реализующая задачи-передатчики */
void vSenderTask(void *pvParameters) {
    /* Будет создано несколько экземпляров задачи. В качестве
    * параметра задаче будет передан указатель на структуру xData. */
    /* Переменная, которая будет хранить результат выполнения
    * xQueueSendToBack(): */
    portBASE_TYPE xStatus;

    /* Бесконечный цикл */
    for (;;) {
        /* Записать структуру в конец очереди.
        * 1-й параметр — дескриптор очереди, в которую будет
        * производиться запись, очередь создана до запуска
        * планировщика, и ее дескриптор сохранен в глобальной
        * переменной xQueue.
        * 2-й параметр — указатель на структуру, которая будет
        * записана в очередь, в данном случае указатель передается
        * при создании экземпляра задачи (pvParameters).
        * 3-й параметр — продолжительность тайм-аута, в течение
        * которого задача будет находиться в блокированном
        * состоянии, ожидая появления свободного места в очереди.
        * Макроопределение portTICK_RATE_MS используется для
        * преобразования времени 100 мс в количество системных
        * квантов. */
        xStatus = xQueueSendToBack(xQueue, pvParameters, 100 /
portTICK_RATE_MS);
        if (xStatus != pdPASS) {
            /* Запись в очередь не произошла по причине того, что
            * очередь на протяжении тайм-аута оставалась заполненной.
            * Такая ситуация свидетельствует об ошибке, так как
            * очередь-приемник создаст свободное место в очереди,
            * как только обе задачи-передатчика перейдут
            * в блокированное состояние */
            puts("Could not send to the queue.\r\n");
        }
        /* Сделать принудительный вызов планировщика, позволив,
        * таким образом, выполняться другой задаче-передатчику.
        * Переключение на другую задачу произойдет быстрее, чем
        * окончится текущий квант времени. */
        taskYIELD();
    }
}

/*-----------------------------------------------------------*/
/* Функция, реализующая задачу-приемник */
void vReceiverTask(void *pvParameters) {
    /* Структура, в которую будет копироваться прочитанная из
     * очереди структура */
 xData xReceivedStructure;
    /* Переменная, которая будет хранить результат выполнения
     * xQueueReceive() */
    portBASE_TYPE xStatus;
    /* Бесконечный цикл */
    for (;;) {
        /* Эта задача выполняется, только когда задачи-передатчики
        * находятся в блокированном состоянии, а за счет того, что
        * приоритет у них выше, блокироваться они могут, только
        * если очередь полна. Поэтому очередь в этот момент должна
        * быть полна. То есть текущее количество элементов
        * очереди должно быть равно ее размеру — 3. */
        if (uxQueueMessagesWaiting(xQueue) != 3) {
            puts("Queue should have been full!\r\n");
        }
        /* Прочитать структуру из начала очереди.
         * 1-й параметр — дескриптор очереди, из которой будет
        * происходить чтение, очередь создана до запуска
        * планировщика, и ее дескриптор сохранен в глобальной
        * переменной xQueue.
        * 2-й параметр — указатель на буфер, в который будет
        * скопирована структура из очереди. В данном случае —
        * указатель на структуру xReceivedStructure.
        * 3-й параметр — продолжительность тайм-аута. В данном
        * случае задана равной 0, что означает задача не будет
        * "ожидать", если очередь пуста. Однако так как эта задача
        * получает управление, только если очередь полна, то чтение
        * элемента из нее будет всегда возможно.
         */
        xStatus = xQueueReceive(xQueue, &xReceivedStructure, 0);
        if (xStatus == pdPASS) {
            /* Структура успешно принята, вывести на экран название
             * задачи, которая эту структуру поместила в очередь,
             * и значение абстрактного параметра */
            if (xReceivedStructure.ucSource == mainSENDER_1) {
                printf("From Sender 1 = %d\r\n", xReceivedStructure.ucValue);
            } else {
                printf("From Sender 2 = %d\r\n", xReceivedStructure.ucValue);
            }
        } else {
            /* Данные не были прочитаны из очереди.
             * При условии, что очередь должна быть полна, означает
             * аварийную ситуацию */
            puts("Could not receive from the queue.\r\n");
        }
    }
}

/*-----------------------------------------------------------*/
/* Точка входа. С функции main() начнется выполнение программы. */
int main(void) {
    /* Создать очередь размером 3 элемента для хранения
    * структуры типа xData.
    * Размер элемента установлен равным размеру структуры xData.
    * Дескриптор созданной очереди сохранить в глобальной
    * переменной xQueue. */
    xQueue = xQueueCreate(3, sizeof(xData));
    /* Если очередь успешно создана (дескриптор не равен NULL) */
    if (xQueue != NULL) {
        /* Создать 2 экземпляра задачи-передатчика. Параметр,
        * передаваемый задаче при ее создании, указатель на структуру,
        * которую экземпляр задачи-передатчика
        * будет записывать в очередь.
        * Задача-передатчик 1 будет постоянно записывать структуру
        * xStructsToSend[ 0 ].
        * Задача-передатчик 2 будет постоянно записывать структуру
        * xStructsToSend[ 1 ].
        * Обе задачи создаются с приоритетом 1.
        */
        xTaskCreate(vSenderTask, "Sender1", 1000, ( void * ) &(
xStructsToSend[ 0 ] ), 2, NULL);
        xTaskCreate(vSenderTask, "Sender2", 1000, ( void * ) &(
xStructsToSend[ 1 ] ), 2, NULL);
        /* Создать задачу-приемник, которая будет считывать числа
         * из очереди.
         * Приоритет = 2, то есть выше, чем у задач-передатчиков.
         */
        xTaskCreate(vReceiverTask, "Receiver", 1000, NULL, 1, NULL);
        /* Запуск планировщика. Задачи начнут выполняться. */
        vTaskStartScheduler();
    } else {
        /* Если очередь не создана */
    }
    /* При успешном создании очереди и запуске планировщика
     * программа никогда "не дойдет" до этого места. */
    for (;;)
        ;
}
```

Результат выполнения учебной программы № 3 показан на рис. 6, на котором видно, что теперь задача-приемник владеет информацией о том, какая именно задача передала то или иное сообщение.

В момент времени (1) (рис. 7) управление получает одна из задач-передатчиков, так как приоритет их выше, чем у задачи-приемника. Пусть это будет задача-передатчик 1. Она записывает первый элемент в пустую очередь и вызывает планировщик (момент времени (2)). Планировщик передает управление другой задаче с таким же приоритетом, то есть задаче-передатчику 2. Та записывает еще один элемент в очередь (теперь в очереди 2 элемента) и отдает управление задаче-передатчику 1 (момент времени (3)). Задача-передатчик 1 записывает 3-й элемент в очередь, теперь очередь заполнена. Когда управление передается задаче-передатчику 2, она обнаруживает, что не может записать новый элемент в очередь, и переходит в блокированное состояние (момент времени (5)). Управление снова получает задача-передатчик 1, однако очередь по-прежнему заполнена, и задача-передатчик 1 также блокируется в ожидании освобождения места в очереди (момент времени (6)).

Так как все задачи с приоритетом 2 теперь блокированы, управление получает задача-приемник, приоритет которой ниже и равен «1» (момент времени (6)). Она считывает один элемент из очереди, освобождая таким

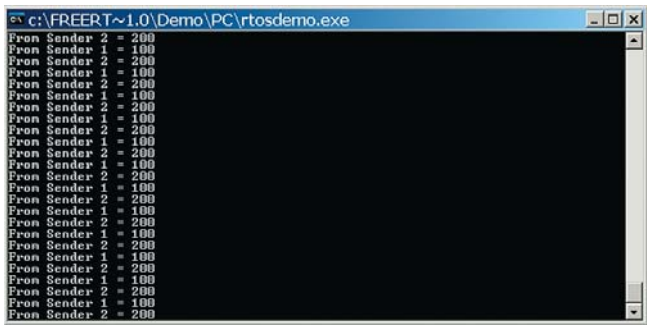

Рис. 6. Результат выполнения учебной программы № 3

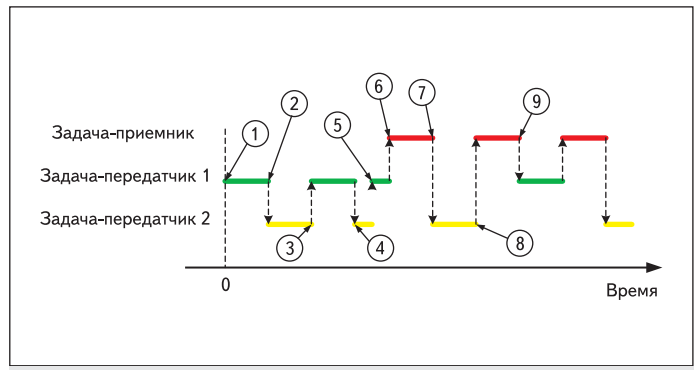

Рис. 7. Последовательность выполнения задач в учебной программе № 3

образом место в очереди. Как только в очереди появилось свободное место, планировщик выведет из состояния блокировки ту задачу из числа «ожидавших», которая дольше остальных пребывала блокированной. В нашем случае это задача-передатчик 2 (момент времени (7)). Так как приоритет у нее выше, она вытеснит задачу-приемник и запишет следующий элемент в очередь. После чего она вызовет планировщик API-функцией ***taskYIELD()***. Однако готовых к выполнению задач с более высоким или равным приоритетом на этот момент нет, поэтому переключения контекста не произойдет, и задача-передатчик 2 продолжит выполняться. Она попытается записать в очередь еще один элемент, но очередь заполнена, и задача-передатчик 2 перейдет в блокированное состояние (момент времени (8)).

Снова сложилась ситуация, когда все высокоприоритетные задачи-передатчики блокированы, поэтому управление получит низкоприоритетная задача-приемник (8). Однако на этот раз после появления свободного места в очереди разблокируется задача-передатчик 1, так как теперь ее время пребывания в блокированном состоянии превышает время задачи-передатчика 2, и т. д.

Следует отметить, что в ранее приведенном примере, когда задачи-передатчики имеют более высокий приоритет, чем задача-приемник, в очереди в любой момент времени не может быть более одного свободного места.

Использование очередей для передачи больших объемов данных

Если размер одного элемента очереди достаточно велик, то предпочтительно использовать очередь для хранения не самих элементов, а для хранения указателей на элементы (например, на массивы или на структуры).

Преимущества такого подхода:

- Экономия памяти. Память при создании очереди выделяется под все элементы очереди, даже если очередь пуста. Использование небольших по объему занимаемой памяти указателей вместо объемных структур или массивов позволяет достичь существенной экономии памяти.
- Меньшее время записи элемента в очередь и чтения его из очереди. При записи/чтении элемента из очереди происходит его побайтовое копирование. Копирование указателя выполняется быстрее копирования объемных структур данных.

Тем не менее использование указателей в качестве элементов очереди сопряжено с некоторыми трудностями, преодоление которых ложится на плечи программиста. Для достижения корректной работы программы должны быть выполнены следующие условия:

- У памяти, адресуемой указателем, в каждый момент времени должна быть одна четко определенная задача-хозяин, которая может обращаться к этой памяти. То есть необходимо гарантировать, что несколько задач не будут одновременно обращаться к памяти, на которую ссылается указатель. В идеальном случае только задача-передатчик должна иметь доступ к памяти, пока указатель на эту память находится в очереди. Когда же указатель прочитан из очереди, только задача-приемник должна иметь возможность доступа к памяти.
- Память, на которую ссылается указатель, должна существовать. Это требование актуально, если указатель ссылается на динамически выделенную память. Только одна задача должна быть ответственна за освобождение динамически выделенной памяти. Задачи не должны обращаться к памяти, если та уже была освобождена.
- Нельзя использовать указатель на переменные, расположенные в стеке задачи, то есть указатель на локальные переменные задачи. Данные, на которые ссылается указатель, будут неверными после очередного переключения контекста.

Выводы

В этой части статьи был подробно описан механизм очередей как средства межзадачного взаимодействия. Показаны основные способы организации такого взаимодействия. Однако существуют еще несколько API-функций для работы с очередями, которые используются только для отладки ядра FreeRTOS. О них будет рассказано в дальнейших публикациях, посвященных возможностям отладки и трассировки. В следующей же публикации внимание будет сконцентрировано на особенностях обработки прерываний микроконтроллера в среде FreeRTOS.

Литература

1. Курниц А. FreeRTOS — операционная система для микроконтроллеров // Компоненты и технологии. 2011. № 2–5.
2. Barry R. Using the FreeRTOS real time kernel. A Practical Guide. 2009.
3. www.freertos.org
4. http://ru.wikipedia.org/wiki/Очередь_(программирование)