Продолжение. Начало в № 2`2011

FreeRTOS — операционная система для микроконтроллеров

Андрей КУРНИЦ
kurnits@stim.by

Автор этой статьи продолжает знакомить читателя с созданием программ, работающих под управлением FreeRTOS — операционной системы для микроконтроллеров. На этот раз речь пойдет об альтернативном способе реализации многозадачной среды, когда в программе вместо задач используются сопрограммы. Мы оценим достоинства и недостатки использования сопрограмм.

Что представляет собой сопрограмма?

В предыдущих публикациях [1] мы говорили о FreeRTOS как о многозадачной операционной системе, и в центре нашего внимания находилась задача (task) — базовая единица программы, работающей под управлением FreeRTOS. Речь шла о том, что программа, работающая под управлением FreeRTOS, разбивается на совокупность задач. Задача представляет собой отдельный поток команд процессора и реализуется в виде функции языка Си. Каждая задача отвечает за небольшую часть функциональности всей программы. Каждая задача выполняется независимо от остальных, но взаимодействует с остальными задачами через механизмы межзадачного взаимодействия.

Начиная с версии v4.0.0 во FreeRTOS появилась поддержка сопрограмм (co-routines). Сопрограмма сходна с задачей, она также представляет собой независимый поток команд процессора, и ее можно использовать как базовую единицу программы. То есть программа, работающая под управлением FreeRTOS, может состоять из совокупности сопрограмм.

Когда следует использовать сопрограммы?

Главное преимущество сопрограмм перед задачами — это то, что использование сопрограмм позволяет достичь значительной экономии оперативной памяти по сравнению с использованием задач.

Каждой задаче для корректной работы ядро выделяет участок памяти, в которой размещаются стек задачи и структура управления задачей (Task Control Block). Размер этого участка памяти за счет размещения в нем стека оказывается значительным. Так как объем оперативной памяти в микроконтроллерах ограничен, то его может оказаться недостаточно для размещения всех задач.

В таких случаях одним из возможных решений будет замена всех (или части) задач на сопрограммы. В этом случае программа будет представлять собой совокупность независимых друг от друга и взаимодействующих друг с другом сопрограмм.

Сопрограммам по сравнению с задачами присущ ряд существенных ограничений, поэтому использование сопрограмм оправдано только в случаях, когда объема оперативной памяти оказывается недостаточно. Следует отметить, что в программе допускается совместное использование как задач, так и сопрограмм.

Особенности сопрограмм:

1. Использование стека. Все сопрограммы в программе используют один и тот же стек, это позволяет добиться значительной экономии оперативной памяти по сравнению с использованием задач, но налагает ряд ограничений при программировании сопрограмм.
2. Планирование и приоритеты. Сопрограммы в отличие от задач выполняются в режиме кооперативной многозадачности с приоритетами. Кооперативная многозадачность в отношении сопрограмм автоматически устраняет проблему реентерабельности функций, но негативно сказывается на времени реакции микроконтроллерной системы на внешние события.
3. Сочетание с задачами. Сопрограммы могут выполняться одновременно с задачами, которые обслуживаются планировщиком с вытесняющей многозадачностью. При этом задачи выполняются в первую очередь, и только если нет готовых к выполнению задач, процессор занят выполнением сопрограмм. Важно, что во FreeRTOS не существует встроенного механизма взаимодействия между задачами и сопрограммами.
4. Примитивность. По сравнению с задачами сопрограммы не допускают целый ряд операций.
   – Операции с семафорами и мьютексами не представлены для сопрограмм.
   – Набор операций с очередями ограничен по сравнению с набором операций для задач.
   – Сопрограмму после создания нельзя уничтожить или изменить ее приоритет.
5. Ограничения в использовании:
   – Внутри сопрограмм нельзя использовать локальные переменные.
   – Существуют строгие требования к месту вызова API-функций внутри сопрограмм.

Экономия оперативной памяти при использовании сопрограмм

Оценим объем оперативной памяти, который можно сэкономить, применяя сопрограммы вместо задач.

Пусть в качестве платформы выбран микроконтроллер семейства AVR. Настройки ядра FreeRTOS идентичны настройкам демонстрационного проекта, который входит в дистрибутив FreeRTOS. Рассмотрим два случая. В первом случае вся функциональность программы реализована десятью задачами, во втором — десятью сопрограммами.

Оперативная память, потребляемая одной задачей, складывается из памяти стека и памяти, занимаемой блоком управления задачей. Для условий, приведенных выше, размер блока управления задачей составляет 33 байт, а рекомендованный минимальный размер стека — 85 байт. Таким образом, имеем 33+85 = 118 байт на каждую задачу. Для создания 10 задач потребуется 1180 байт.

Оперативная память, потребляемая одной сопрограммой, складывается только из памяти, занимаемой блоком управления сопрограммой. Размер блока управления сопрограммой для данных условий равен 26 байт. Как упоминалось выше, стек для всех сопрограмм общий, примем его равным рекомендованному, то есть 85 байт. Для создания 10 сопрограмм потребуется 10×26+85 = 345 байт.

Таким образом, используя сопрограммы, удалось достичь экономии оперативной памяти 1180–345 = 835 байт, что составляет приблизительно 71%.

Состояния сопрограммы

Как и задача, сопрограмма может пребывать в одном из нескольких возможных состояний. Для сопрограмм этих состояний три:

1. Выполнение (Running). Говорят, что сопрограмма выполняется, когда в данный момент времени процессор занят непосредственно ее выполнением. В любой момент времени только одна сопрограмма в системе может находиться в состоянии выполнения.
2. Готовность к выполнению (Ready). Говорят, что сопрограмма готова к выполнению, если она не блокирована, однако в данный момент процессор занят выполнением другой сопрограммы или какой-то задачи. Сопрограмма может находиться в состоянии готовности к выполнению по одной из следующих причин:
   – Другая сопрограмма в данный момент находится в состоянии выполнения.
   – Одна из задач находится в состоянии выполнения, если в программе одновременно используются и сопрограммы, и задачи.
3. Блокированное состояние (Blocked). Сопрограмма блокирована, когда ожидает наступления некоторого события. Как и в случае с задачами, событие может быть связано с отсчетом заданного временного интервала — временное событие, а может быть связано с ожиданием внешнего по отношению к сопрограмме события. Например, если сопрограмма вызовет API-функцию ***crDELAY()***, то она перейдет в блокированное состояние и пробудет в нем на протяжении заданного интервала времени. Блокированные сопрограммы не получают процессорного времени.

Графически состояния сопрограммы и переходы между ними представлены на рис. 1. В отличие от задач у сопрограмм нет приостановленного (suspended) состояния, однако оно может быть добавлено в будущих версиях FreeRTOS.

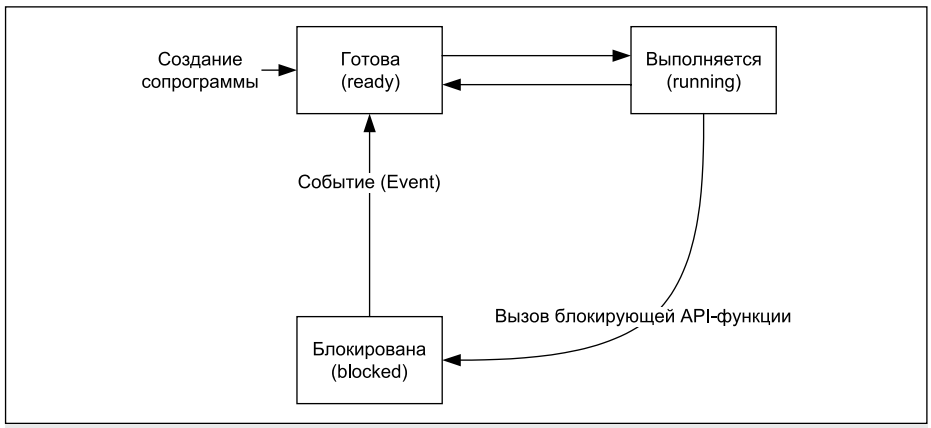

Рис. 1. Состояния сопрограммы

Выполнение сопрограмм и их приоритеты

Как и при создании задачи, при создании сопрограммы ей назначается приоритет. Сопрограмма с высоким приоритетом имеет преимущество на выполнение перед сопрограммой с низким приоритетом.

Следует помнить, что приоритет сопрограммы дает преимущество на выполнение одной сопрограммы только перед другой сопрограммой. Если в программе используются как задачи, так и сопрограммы, то задачи всегда будут иметь преимущество перед сопрограммами. Сопрограммы выполняются только тогда, когда нет готовых к выполнению задач.

Важно, что преимущество на выполнение не означает, что если в системе появилась готовая к выполнению сопрограмма с более высоким приоритетом, чем та, что выполняется в данный момент, то управление получит эта высокоприоритетная сопрограмма.

Сопрограммы выполняются в режиме кооперативной многозадачности. Это означает, что одна сопрограмма сменяет другую лишь тогда, когда выполняющаяся в данный момент сопрограмма сама передает управление другой сопрограмме посредством вызова API-функции. Причем если в момент передачи управления в состоянии готовности к выполнению находятся несколько сопрограмм, то управление получит самая высокоприоритетная среди них.

Итак, сопрограмма прерывает свое выполнение только при выполнении одного из следующих условий:

1. Сопрограмма перешла в блокированное состояние, вызвав соответствующую API-функцию.
2. Сопрограмма выполнила принудительное переключение на другую сопрограмму (аналог принудительного переключения контекста задачи).
3. Сопрограмма была вытеснена задачей, которая до этого находилась в приостановленном или блокированном состоянии.

Сопрограмма не может быть вытеснена другой сопрограммой, однако появившаяся готовая к выполнению задача вытесняет любую сопрограмму.

Для корректного выполнения сопрограмм необходимо организовать в программе периодический вызов API-функции ***vCoRoutineSchedule()***. Рекомендованное место для вызова API-функции ***vCoRoutineSchedule()*** — тело задачи Бездействие, подробнее об этом будет написано ниже. После первого вызова ***vCoRoutineSchedule()*** управление получает сопрограмма с наивысшим приоритетом.

Приоритет сопрограммы задается целым числом, которое может принимать значения от 0 до (***configMAX_CO_ROUTINE_PRIORITIES*** — 1). Большее значение соответствует более высокому приоритету. Макроопределение ***configMAX_CO_ROUTINE_PRIORITIES*** задает общее число приоритетов сопрограмм в программе и определено в конфигурационном файле ***FreeRTOSConfig.h***. Изменяя значение ***configMAX_CO_ROUTINE_PRIORITIES***, можно определить любое число возможных приоритетов сопрограмм, однако следует стремиться уменьшить число приоритетов до минимально достаточного для экономии оперативной памяти, потребляемой ядром.

Реализация сопрограммы

Как и задача, сопрограмма реализуется в виде функции языка Си. Указатель на эту функцию следует передавать в качестве аргумента API-функции создания сопрограммы, о которой будет сказано ниже. Пример функции, реализующей сопрограмму:

```
void vACoRoutineFunction(xCoRoutineHandle xHandle, unsigned
portBASE_TYPE uxIndex)
{
  crSTART( xHandle );

  for(;; )
  {
    // Код, реализующий функциональность сопрограммы,
       размещается здесь.
  }

  crEND();
}
```

Аргументы функции, реализующей сопрограмму:

1. ***xHandle*** — дескриптор сопрограммы. Автоматически передается в функцию, реализующую сопрограмму, и в дальнейшем используется при вызове API-функций для работы с сопрограммами.
2. ***uxIndex*** — произвольный целочисленный параметр, который передается в сопрограмму при ее создании.

Указатель на функцию, реализующую сопрограмму, определен в виде макроопределения ***crCOROUTINE_CODE***.

К функциям, реализующим сопрограммы, предъявляются следующие требования:
1. Функция должна начинаться с вызова API-функции ***crSTART()***.
2. Функция должна завершаться вызовом API-функции ***crEND()***.
3. Как и в случае с задачей, функция никогда не должна заканчивать свое выполнение, весь полезный код сопрограммы должен быть заключен внутри бесконечного цикла.
4. Сопрограммы выполняются в режиме кооперативной многозадачности. Поэтому если в программе используется несколько сопрограмм, то для того, чтобы процессорное время получали все сопрограммы в программе, бесконечный цикл должен содержать вызовы блокирующих API-функций.

Создание сопрограммы

Для создания сопрограммы следует до запуска планировщика вызвать API-функцию ***xCoRoutineCreate()***, прототип которой приведен ниже:

```
portBASE_TYPE xCoRoutineCreate(
                        crCOROUTINE_CODE pxCoRoutineCode,
                        unsigned portBASE_TYPE uxPriority,
                        unsigned portBASE_TYPE uxIndex
                        );
```

Аргументы и возвращаемое значение:
1. ***pxCoRoutineCode*** — указатель на функцию, реализующую сопрограмму (фактически — идентификатор функции в программе).
2. ***uxPriority*** — приоритет создаваемой сопрограммы. Если задано значение больше, чем (***configMAX_CO_ROUTINE_PRIORITIES*** — 1), то сопрограмма получит приоритет? равный (***configMAX_CO_ROUTINE_PRIORITIES*** — 1).
3. ***uxIndex*** — целочисленный параметр, который передается сопрограмме при ее создании. Позволяет создавать несколько экземпляров одной сопрограммы.
4. Возвращаемое значение. Равно ***pdPASS***, если сопрограмма успешно создана и добавлена к списку готовых к выполнению, в противном случае — код ошибки, определенный в файле ***ProjDefs.h*** (обычно ***errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY***).

API-функция vCoRoutineSchedule()

Выполнение сопрограмм должно быть организовано при помощи циклического вызова API-функции ***vCoRoutineSchedule()***. Ее прототип:

```
void vCoRoutineSchedule( void );
```

Вызов ***vCoRoutineSchedule()*** рекомендуется располагать в задаче Бездействие:

```
void vApplicationIdleHook( void )
{
 vCoRoutineSchedule( void );
}
```

Если задача Бездействие не выполняет никаких других функций, то более эффективной будет следующая ее реализация:

```
void vApplicationIdleHook( void )
{
   for( ;; )
   {
      vCoRoutineSchedule( void );
   }
}
```

Даже если в программе не используется ни одной задачи, задача Бездействие автоматически создается при запуске планировщика.

Вызов API-функции ***vCoRoutineSchedule()*** внутри задачи Бездействие позволяет легко сочетать в одной программе как задачи, так и сопрограммы. При этом сопрограммы будут выполняться, только если нет готовых к выполнению задач с приоритетом выше приоритета задачи Бездействие (который обычно равен 0).

В принципе вызов API-функции ***vCoRoutineSchedule()*** возможен в любой задаче, а не только в задаче Бездействие. Обязательным требованием является то, чтобы задача, из которой вызывается ***vCoRoutineSchedule()***, имела самый низкий приоритет. Иначе если существуют задачи с более низким приоритетом, то они не будут получать процессорное время.

Важно, что стек, общий для всех сопрограмм, является стеком той задачи, которая вызывает API-функцию ***vCoRoutineSchedule()***. Если вызов ***vCoRoutineSchedule()*** располагается в теле задачи Бездействие, то все сопрограммы используют стек задачи Бездействие. Размер стека задачи Бездействие задается макроопределением ***configMINIMAL_STACK_SIZE*** в файле ***FreeRTOSConfig.h***.

Настройки FreeRTOS для использования сопрограмм

Для того чтобы организовать многозадачную среду на основе сопрограмм, прежде всего необходимо соответствующим образом настроить ядро FreeRTOS:
1. В исходный текст программы должен быть включен заголовочный файл croutine.h, содержащий определения API-функций для работы с сопрограммами:

```
#include "croutine.h"
```

2. Конфигурационный файл ***FreeRTOSConfig.h*** должен содержать следующие макроопределения, установленные в 1: ***configUSE_IDLE_HOOK*** и ***configUSE_CO_ROUTINES***.
3. Следует также определить количество приоритетов сопрограмм. Файл ***FreeRTOSConfig.h*** должен содержать макроопределение вида:

```
#define configMAX_CO_ROUTINE_PRIORITIES ( 3
```

Учебная программа № 1

Рассмотрим учебную программу № 1, в которой создаются 2 сопрограммы и реализовано их совместное выполнение. Каждая сопрограмма сигнализирует о своем выполнении, после чего реализуется временная задержка с помощью пустого цикла, далее происходит принудительное переключение на другую сопрограмму. Приоритет сопрограмм установлен одинаковым.

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

/* Функция, реализующая Сопрограмму 1.
 Параметр, передаваемый в сопрограмму при ее создании,
 не используется.  Сопрограмма сигнализирует о своем
 выполнении, после чего блокируется на 500 мс. */
void vCoRoutine1( xCoRoutineHandle xHandle, unsigned portBASE_
TYPE uxIndex ) {
   /* Все переменные должны быть объявлены как static. */
   static long i;
   /* Сопрограмма должна начинаться с вызова crSTART().
   Дескриптор сопрограммы xHandle получен автоматически
   в виде аргумента функции, реализующей сопрограмму
   vCoRoutine1().*/
   crSTART( xHandle );
   /* Сопрограмма должна содержать бесконечный цикл. */
   for(;;) {
      /* Сигнализировать о выполнении */
      puts("Co-routine #1 runs!");
      /* Пауза, реализованная с помощью пустого цикла */
      for (i = 0; i < 5000000; i++);
      /* Выполнить принудительное переключение на другую со-
программу */
      crDELAY( xHandle, 0 );
   }
   /* Сопрограмма должна завершаться вызовом crEND(). */
   crEND();
}

/* Функция, реализующая Сопрограмму 2.
 Сопрограмма 2 выполняет те же действия, что и Сопрограмма 1.*/
void vCoRoutine2( xCoRoutineHandle xHandle, unsigned portBASE_
TYPE uxIndex ) {
   static long i;
   crSTART( xHandle );
   for(;;) {
      /* Сигнализировать о выполнении */
      puts("Co-routine #2 runs!");
      /* Пауза, реализованная с помощью пустого цикла */
      for (i = 0; i < 5000000; i++);
      /* Выполнить принудительное переключение на другую со-
программу */
      crDELAY( xHandle, 0 );
   }
   crEND();
}

/* Точка входа. С функции main() начинается выполнение про-
граммы. */
void main(void) {

   /* До запуска планировщика создать Сопрограмму 1
и Сопрограмму 2.
   Приоритеты сопрограмм одинаковы и равны 1.
   Параметр, передаваемый при создании, не используется и ра-
вен 0. */
   xCoRoutineCreate(vCoRoutine1, 1, 0);
   xCoRoutineCreate(vCoRoutine2, 1, 0);

   /* В программе не создается ни одной задачи.
   Однако задачи можно добавить, создавая их до запуска плани-
ровщика */

   /* Запуск планировщика. Сопрограммы начнут выполняться.
*/
   vTaskStartScheduler();
}

/* Функция, реализующая задачу Бездействие, должна присутство-
вать в программе и содержать вызов vCoRoutineSchedule() */
```

Рис. 2. Результат выполнения учебной программы № 1

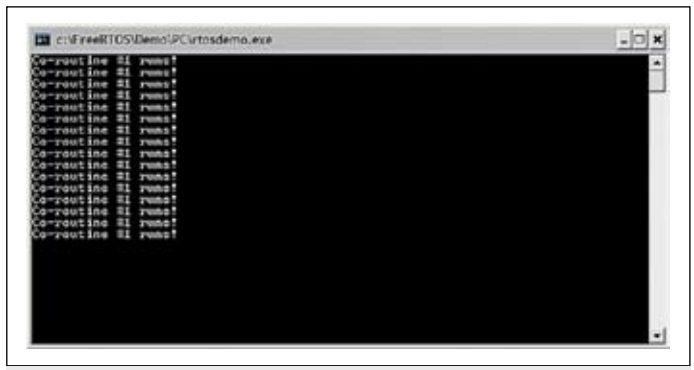

Рис. 4. Результат выполнения учебной программы № 1, когда Сопрограмма 1 не выполняет переключения на другую сопрограмму

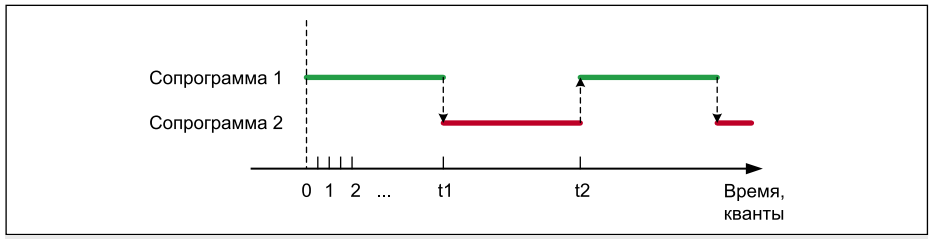

Рис. 3. Ход выполнения сопрограмм в учебной программе № 1

```
void vApplicationIdleHook(void) {
   /* Так как задача Бездействие не выполняет других действий,
   то вызов vCoRoutineSchedule() размещен внутри бесконечного
цикла.*/
   for (;;) {
      vCoRoutineSchedule();
   }
}
```

Результаты работы учебной программы № 1 приведены на рис. 2. На рис. 2 видно, что сообщения на дисплей выводят обе сопрограммы, следовательно, каждая из них получает процессорное время. На рис. 3 представлено разделение процессорного времени между сопрограммами.

Сопрограммы выполняются в режиме кооперативной многозадачности, поэтому текущая сопрограмма выполняется до тех пор, пока не произойдет явное переключение на другую сопрограмму. На протяжении времени 0…t1 будет выполняться только Сопрограмма 1, а именно будет выполняться продолжительный по времени пустой цикл (рис. 3). Как только пустой цикл Сопрограммы 1 будет завершен, в момент времени t1 произойдет явное переключение на другую сопрограмму. В результате чего управление получит Сопрограмма 2 на такой же продолжительный промежуток времени — t1…t2.

Следует обратить внимание на обязательный вызов API-функции ***crDELAY(xHandle, 0)***, благодаря которому происходит принудительное переключение на другую сопрограмму и, таким образом, реализуется принцип кооперативной многозадачности.

Продемонстрировать важность «ручного» переключения на другую сопрограмму можно, если исключить из функции Сопрограммы 1 вызов API-функции ***crDELAY()***. В таком случае результаты работы программы (рис. 4) будут свидетельствовать о том, что процессорное время получает только Сопрограмма 1. Причиной этому является тот факт, что Сопрограмма 1 не выполняет принудительного переключения на другую сопрограмму, что является необходимым условием корректной работы кооперативной многозадачной среды.

Ограничения при использовании сопрограмм

Платой за уменьшение объема потребляемой оперативной памяти при использовании сопрограмм вместо задач является то, что программирование сопрограмм сопряжено с рядом ограничений. В целом реализация сопрограмм сложнее, чем реализация задач.

Использование локальных переменных

Особенность сопрограмм в том, что когда сопрограмма переходит в блокированное состояние, стек сопрограммы не сохраняется. То есть если переменная находилась в стеке в момент, когда сопрограмма перешла в блокированное состояние, то по выходу из него значение переменной, вероятно, будет другим. Эта особенность объясняется тем фактом, что все сопрограммы в программе используют один и тот же стек.

Чтобы избежать потери значения переменных, не следует размещать их в стеке, то есть нельзя использовать локальные переменные в сопрограммах. Все переменные, используемые в сопрограмме, должны быть глобальными либо объявлены статическими (ключевое слово static). Рассмотрим пример функции, реализующей сопрограмму:

```
// Глобальная переменная:
unsigned int uGlobalVar;

// Функция, реализующая сопрограмму
void vACoRoutineFunction( xCoRoutineHandle xHandle, unsigned
portBASE_TYPE uxIndex )
{
   // Статическая переменная:
   static unsigned int uStaticVar;

   // Локальная переменная — В СТЕКЕ!!!
   unsigned int uLocalVar = 10L;

   crSTART( xHandle );
   for(;; )
   {
      uGlobalVar = 1;
      uStaticVar = 10;
      uLocalVar = 100;

      // Вызов блокирующей API-функции
      crDELAY( xHandle, 10 );

      // После вызова блокирующей API-функции
      // значение глобальной и статической переменной
      // uGlobalVar и uStaticVar гарантированно сохранится.

      // Значение же локальной переменной uLocalVar
      // может оказаться не равным 100!!!
   }
   crEND();
}
```

Вызов блокирующих API-функций

Еще одним последствием использования общего для всех сопрограмм стека является то, что вызов блокирующих API-функций допускается только непосредственно из тела сопрограммы, но не допускается из функций, которые вызываются из тела сопрограммы. Рассмотрим пример:

```
// Функция, реализующая сопрограмму
void vACoRoutineFunction(xCoRoutineHandle xHandle, unsigned
portBASE_TYPE uxIndex)
{
   crSTART( xHandle );

   for(;; )
   {
      // Непосредственно в сопрограмме
      // блокирующие API-функции вызывать можно.
      crDELAY( xHandle, 10 );

      // Однако внутри функции vACalledFunction() их НЕЛЬЗЯ
вызывать!!!
      vACalledFunction();
   }
   crEND();
}

void vACalledFunction(void) {
   // Здесь нельзя вызывать блокирующие API-функции!!!

   // ОШИБКА!
      crDELAY( xHandle, 10 );
}
```

Внутренняя реализация сопрограмм не допускает вызова блокирующих API-функций внутри выражения ***switch***. Рассмотрим пример:

```
// Функция, реализующая сопрограмму
void vACoRoutineFunction( xCoRoutineHandle xHandle, unsigned
portBASE_TYPE uxIndex )
{
   crSTART( xHandle );

   for(;; )
   {
      // Непосредственно в сопрограмме
      // блокирующие API-функции вызывать можно.
      crDELAY( xHandle, 10 );

      switch( aVariable )
      {
         case 1 : // Здесь нельзя вызывать блокирующие API-функции.
         break;
         default: // Здесь тоже нельзя.
      }
   }
   crEND();
}
```

API-функции, предназначенные для вызова из сопрограмм

Текущая версия FreeRTOS v7.0.1 поддерживает следующие API-функции, предназначенные для вызова из сопрограмм:

- ***crDELAY()***;
- ***crQUEUE_SEND()***;
- ***crQUEUE_RECEIVE()***.

Кроме этого, существуют еще API-функции ***crQUEUE_SEND_FROM_ISR()*** и ***crQUEUE_RECEIVE_FROM_ISR()***, предназначенные для вызова из обработчиков прерываний и выполняющие операции с очередью, которая используется только в сопрограммах.

Все вышеперечисленные API-функции на самом деле представляют собой макросы языка Си, но для простоты будем называть их API-функциями.

Стоит подчеркнуть, что API-функции, предназначенные для вызова из сопрограмм, разрешено вызывать только непосредственно из тела сопрограммы. Вызов их из других функций запрещен.

Префикс всех вышеперечисленных API-функций указывает на заголовочный файл ***croutine.h***, в котором эти API-функции объявлены.

Реализация задержек в сопрограммах

Для корректной реализации временных задержек внутри сопрограмм следует применять API-функцию ***crDELAY()***, которая переводит вызывающую сопрограмму в блокированное состояние на заданное количество квантов времени. Ее прототип:

```
void crDELAY( xCoRoutineHandle xHandle, portTickType
xTicksToDelay );
```

Аргументы API-функции ***crDELAY()***:

1. ***xHandle*** — дескриптор вызывающей сопрограммы. Автоматически передается в функцию, реализующую сопрограмму, в виде первого ее аргумента.
2. ***xTicksToDelay*** — количество квантов времени, в течение которых сопрограмма будет заблокирована. Если ***xTicksToDelay*** равен 0, то вместо блокировки сопрограммы происходит переключение на другую готовую к выполнению сопрограмму.

Отдельно следует обратить внимание на вызов ***crDELAY()***, когда аргумент ***xTicksToDelay*** равен 0. В этом случае вызывающая ***crDELAY(xHandle, 0)*** сопрограмма переходит в состояние готовности к выполнению, а в состояние выполнения переходит другая сопрограмма, приоритет которой выше или равен приоритету вызывающей сопрограммы.

Посредством вызова ***crDELAY(xHandle, 0)*** происходит принудительное переключение на другую сопрограмму, что было продемонстрировано в учебной программе № 1.

Следует отметить, что применительно к сопрограммам не существует аналога API-функции ***vTaskDelayUntil()***, которая предназначена для вызова из задач и позволяет организовать циклическое выполнение какого-либо действия со строго заданным периодом. Также отсутствует аналог API-функции ***xTaskGetTickCount()***, которая позволяет получить текущее значение счетчика квантов.

Использование очередей в сопрограммах

Как известно, очереди во FreeRTOS представляют собой базовый механизм межзадачного взаимодействия, на механизме очередей основываются такие объекты ядра, как семафоры и мьютексы.

FreeRTOS допускает использование очередей и в сопрограммах, но в этом случае существует одно серьезное ограничение: одну и ту же очередь нельзя использовать для передачи сообщений от очереди к сопрограмме и наоборот. Допускается лишь передача сообщений между сопрограммами и обработчиками прерываний. Когда очередь создана, ее следует использовать только в задачах или только в сопрограммах. Эта особенность существенно ограничивает возможности совместного использования задач и сопрограмм.

Следует учитывать, что для сопрограмм набор API-функций для работы с очередями гораздо беднее набора API-функций для задач. Для сопрограмм нет аналогов следующих API-функций:

1) ***uxQueueMessagesWaiting()*** — получение количества элементов в очереди.
2) ***xQueueSendToFront()*** — запись элемента в начало очереди.
3) ***xQueuePeek()*** — чтение элемента из очереди без удаления его из очереди.
4) ***xQueueSendToFrontFromISR()*** — запись элемента в начало очереди из обработчика прерывания.

Запись элемента в очередь

Для записи элемента в очередь из тела сопрограммы служит API-функция ***crQUEUE_SEND()***. Ее прототип:

```
crQUEUE_SEND(
                  xCoRoutineHandle xHandle,
                  xQueueHandle pxQueue,
                  void *pvItemToQueue,
                  portTickType xTicksToWait,
                  portBASE_TYPE *pxResult
                  )
```

Аргументы API-функции ***crQUEUE_SEND()***:

1. ***xHandle*** — дескриптор вызывающей сопрограммы. Автоматически передается в функцию, реализующую сопрограмму, в виде первого ее аргумента.
2. ***pxQueue*** — дескриптор очереди, в которую будет записан элемент. Дескриптор очереди может быть получен при ее создании API-функцией ***xQueueCreate()***.
3. ***pvItemToQueue*** — указатель на элемент, который будет записан в очередь. Размер элемента зафиксирован при создании очереди. Именно это количество байт будет скопировано с адреса, на который ссылается указатель pvItemToQueue.
4. ***xTicksToWait*** — максимальное количество квантов времени, в течение которого сопрограмма может пребывать в блокированном состоянии, если очередь полна и записать новый элемент нет возможности. Для представления времени в миллисекундах следует использовать макроопределение ***portTICK_RATE_MS*** [1, № 4]. Задание ***xTicksToWait*** равным 0 приведет к тому, что сопрограмма не перейдет в блокированное состояние, если очередь полна, и управление будет возвращено сразу же.
5. ***pxResult*** — указатель на переменную типа ***portBASE_TYPE***, в которую будет помещен результат выполнения API-функции ***crQUEUE_SEND()***. Может принимать следующие значения:
   - ***pdPASS*** — означает, что данные успешно записаны в очередь. Если определено время тайм-аута (параметр ***xTicksToWait*** не равен 0), то возврат значения ***pdPASS*** говорит о том, что свободное место в очереди появилось до истечения времени тайм-аута и элемент был помещен в очередь.
   - Код ошибки ***errQUEUE_FULL***, определенный в файле ***ProjDefs.h***.

Следует отметить, что при записи элемента в очередь из тела сопрограммы нет возможности задать время тайм-аута равным бесконечности, такая возможность есть, только если задача записывает элемент в очередь. Установка аргумента ***xTicksToWait*** равным константе ***portMAX_DELAY*** приведет к переходу сопрограммы в блокированное состояние на конечное время, равное ***portMAX_DELAY*** квантов времени. Это связано с тем, что сопрограмма не может находиться в приостановленном (suspended) состоянии.

Чтение элемента из очереди

Для чтения элемента из очереди служит API-функция ***crQUEUE_RECEIVE()***, которую можно вызывать только из тела сопро-

граммы. Прототип API-функции ***crQUEUE_RECEIVE()***:

```
void crQUEUE_RECEIVE(
                    xCoRoutineHandle xHandle,
                    xQueueHandle pxQueue,
                    void *pvBuffer,
                    portTickType xTicksToWait,
                    portBASE_TYPE *pxResult
                    )
```

Аргументы API-функции ***crQUEUE_RECEIVE()***:

1. ***xHandle*** — дескриптор вызывающей сопрограммы. Автоматически передается в функцию, реализующую сопрограмму, в виде первого ее аргумента.
2. ***pxQueue*** — дескриптор очереди, из которой будет прочитан элемент. Дескриптор очереди может быть получен при ее создании API-функцией ***xQueueCreate()***.
3. ***pvBuffer*** — указатель на область памяти, в которую будет скопирован элемент из очереди. Участок памяти, на которую ссылается указатель, должен быть не меньше размера одного элемента очереди.
4. ***xTicksToWait*** — максимальное количество квантов времени, в течение которого сопрограмма может пребывать в блокированном состоянии, если очередь пуста и считать элемент из очереди нет возможности. Для представления времени в миллисекундах следует использовать макроопределение ***portTICK_RATE_MS*** [1, № 4]. Задание ***xTicksToWait*** равным 0 приведет к тому, что сопрограмма не перейдет в блокированное состояние, если очередь пуста, и управление будет возвращено сразу же.
5. ***pxResult*** — указатель на переменную типа ***portBASE_TYPE***, в которую будет помещен результат выполнения API-функции ***crQUEUE_RECEIVE()***. Может принимать следующие значения:
   - ***pdPASS*** — означает, что данные успешно прочитаны из очереди. Если определено время тайм-аута (параметр ***xTicksToWait*** не равен 0), то возврат значения ***pdPASS*** говорит о том, что новый элемент в очереди появился до истечения времени тайм-аута.
   - Код ошибки ***errQUEUE_FULL***, определенный в файле ***ProjDefs.h***.

Как и при записи элемента в очередь из тела сопрограммы, при чтении элемента из очереди также нет возможности заблокировать сопрограмму на бесконечный промежуток времени.

Запись/чтение в очередь из обработчика прерывания

Для организации обмена между обработчиками прерываний и сопрограммами предназначены API-функции ***crQUEUE_SEND_FROM_ISR()*** и ***crQUEUE_RECEIVE_FROM_ISR()***, вызывать которые можно только из обработчиков прерываний. Причем очередь можно использовать только в сопрограммах (но не в задачах).

Запись элемента в очередь (которая используется только в сопрограммах) из обработчика прерывания осуществляется с помощью API-функции ***crQUEUE_SEND_FROM_ISR()***. Ее прототип:

```
portBASE_TYPE crQUEUE_SEND_FROM_ISR(
                    xQueueHandle pxQueue,
                    void *pvItemToQueue,
                    portBASE_TYPE xCoRoutinePreviouslyWoken
                    )
```

Ее аргументы и возвращаемое значение:

1. ***pxQueue*** — 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. Именно это количество байт будет скопировано с адреса, на который ссылается указатель ***pvItemToQueue***.
3. ***xCoRoutinePreviouslyWoken*** — этот аргумент необходимо устанавливать в ***pdFALSE***, если вызов API-функции ***crQUEUE_SEND_FROM_ISR()*** является первым в обработчике прерывания. Если же в обработчике прерывания происходит несколько вызовов ***crQUEUE_SEND_FROM_ISR()*** (несколько элементов помещается в очередь), то аргумент ***xCoRoutinePreviouslyWoken*** следует устанавливать в значение, которое было возвращено предыдущим вызовом ***crQUEUE_SEND_FROM_ISR()***. Этот аргумент введен для того, чтобы в случае, когда несколько сопрограмм ожидают появления данных в очереди, только одна из них выходила из блокированного состояния.
4. Возвращаемое значение. Равно ***pdTRUE***, если в результате записи элемента в очередь разблокировалась одна из сопрограмм. В этом случае необходимо выполнить переключение на другую сопрограмму после выполнения обработчика прерывания.

Чтение элемента из очереди (которая используется только в сопрограммах) из обработчика прерывания осуществляется с помощью API-функции ***crQUEUE_RECEIVE_FROM_ISR()***. Ее прототип:

```
portBASE_TYPE crQUEUE_RECEIVE_FROM_ISR(
                    xQueueHandle pxQueue,
                    void *pvBuffer,
                    portBASE_TYPE * pxCoRoutineWoken
                    )
```

Аргументы и возвращаемое значение:

1. ***pxQueue*** — дескриптор очереди, в которую будет записан элемент. Дескриптор очереди может быть получен при ее создании API-функцией ***xQueueCreate()***.
2. ***pvItemToQueue*** — 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.
3. ***pxCoRoutineWoken*** — указатель на переменную, которая в результате вызова ***crQUEUE_RECEIVE_FROM_ISR()*** примет значение ***pdTRUE***, если одна или несколько сопрограмм ожидали возможности поместить элемент в очередь и теперь разблокировались. Если таковых сопрограмм нет, то значение ****pxCoRoutineWoken*** останется без изменений.
4. Возвращаемое значение:
   - ***pdTRUE***, если элемент был успешно прочитан из очереди;
   - ***pdFALSE*** — в противном случае.

Учебная программа № 2

Рассмотрим учебную программу № 2, в которой продемонстрирован обмен информацией между сопрограммами и обработчиками прерываний. В программе имеются 2 обработчика прерывания и 2 сопрограммы, которые обмениваются друг с другом сообщениями, помещая их в очередь (в программе созданы 3 очереди). В демонстрационных целях в качестве прерываний используются программные прерывания MS-DOS, а служебная сопрограмма выполняет вызов этих прерываний.

В графическом виде обмен информацией в учебной программе № 2 показан на рис. 5.

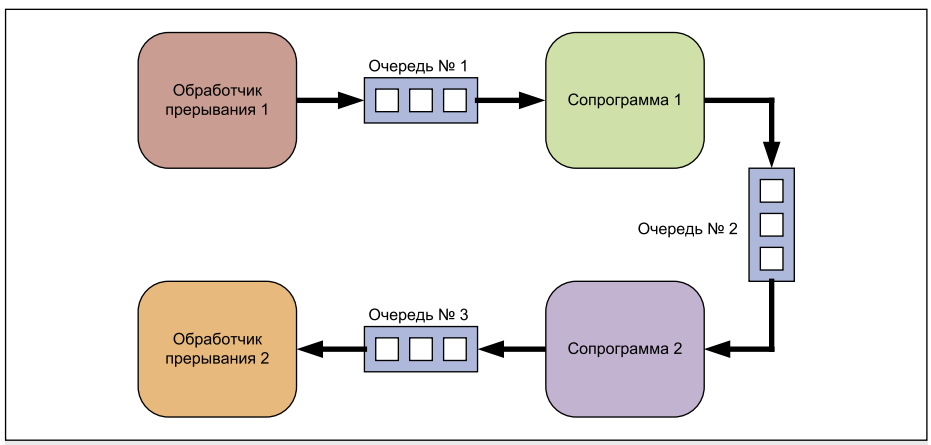

Рис. 5. Обмен сообщениями между сопрограммами и обработчиками прерываний в учебной программе № 2

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"
#include "croutine.h"

/* Дескрипторы очередей — глобальные переменные */
xQueueHandle xQueue1;
xQueueHandle xQueue2;
xQueueHandle xQueue3;

/* Служебная сопрограмма. Вызывает программные прерывания.
 * Приоритет = 1.*/
void vIntCoRoutine( xCoRoutineHandle xHandle, unsigned
portBASE_TYPE uxIndex ) {
   crSTART( xHandle );
   for(;;) {
      /* Эта инструкция сгенерирует прерывание № 1. */
      __asm {int 0x83}
      /* Заблокировать сопрограмму на 500 мс */
      crDELAY(xHandle, 500);
      /* Эта инструкция сгенерирует прерывание № 2. */
      __asm {int 0x82}
      /* Заблокировать сопрограмму на 500 мс */
      crDELAY(xHandle, 500);
   }
   crEND();
}
/*-----------------------------------------------------------*/

/* Функция, реализующая Сопрограмму № 1 и Сопрограмму № 2,
 * то есть будет создано два экземпляра этой сопрограммы. */
void vTransferCoRoutine( xCoRoutineHandle xHandle, unsigned
portBASE_TYPE uxIndex ) {
   static long i;
   portBASE_TYPE xResult;

   crSTART( xHandle );
   for(;;) {
      /* Если выполняется Сопрограмма № 1*/
      if (uxIndex == 1) {
         /* Получить сообщение из Очереди № 1 от Прерывания № 1.
          * Если очередь пуста — заблокироваться на время
portMAX_DELAY квантов */
         crQUEUE_RECEIVE(
               xHandle,


               xQueue1,
               (void *)&i,
               portMAX_DELAY,
               &xResult);
         if (xResult == pdTRUE) {
            puts("CoRoutine 1 has received a message from Interrupt 1.");
         }
         /* Передать это же сообщение в Очередь № 2 Сопрограмме № 2 */
         crQUEUE_SEND(
               xHandle,
               xQueue2,
               (void *)&i,
               portMAX_DELAY ,
               &xResult);
         if (xResult == pdTRUE) {
            puts("CoRoutine 1 has sent a message to CoRoutine 2.");
         }
      }
      /* Если выполняется Сопрограмма № 2 */
      else if (uxIndex == 2) {
         /* Получить сообщение из Очереди № 2 от Сопрограммы № 1.
          * Если очередь пуста — заблокироваться на время
portMAX_DELAY квантов. */
         crQUEUE_RECEIVE(
               xHandle,
               xQueue2,
               (void *)&i,
               portMAX_DELAY,
               &xResult);
         if (xResult == pdTRUE) {
            puts("CoRoutine 2 has received a message from CoRoutine 1.");
         }
         /* Передать это же сообщение в обработчик прерывания
№ 2 через Очередь № 3. */
         crQUEUE_SEND(
               xHandle,
               xQueue3,
               (void *)&i,
               portMAX_DELAY ,
               &xResult);
         if (xResult == pdTRUE) {
            puts("CoRoutine 2 has sent a message to Interrupt 1.");
         }
      }
   }
   crEND();
}
/*-----------------------------------------------------------*/

/* Обработчик Прерывания 1*/
static void __interrupt __far vSendInterruptHandler( void )
{
   static unsigned long ulNumberToSend;

   if (crQUEUE_SEND_FROM_ISR( xQueue1,
         &ulNumberToSend,
         pdFALSE ) == pdPASS) {
      puts("Interrupt 1 has sent a message!");
   }
}
/*-----------------------------------------------------------*/

/* Обработчик Прерывания 2*/
static void __interrupt __far vReceiveInterruptHandler( void )
{
   static portBASE_TYPE pxCoRoutineWoken ;
   static unsigned long ulReceivedNumber;

   /* Аргумент API-функции crQUEUE_RECEIVE_FROM_ISR(),
    который устанавливается в pdTRUE,
    если операция с очередью разблокирует более
    высокоприоритетную сопрограмму.
    Перед вызовом crQUEUE_RECEIVE_FROM_ISR()
    следует установить в pdFALSE. */
   pxCoRoutineWoken = pdFALSE;

   if (crQUEUE_RECEIVE_FROM_ISR(
         xQueue3,
         &ulReceivedNumber,
         &pxCoRoutineWoken ) == pdPASS) {
      puts("Interrupt 2 has received a message!\n");
   }

   /* Проверить, нуждается ли в разблокировке более
    * высокоприоритетная сопрограмма,
    * чем та, что была прервана прерыванием. */
   if( pxCoRoutineWoken == pdTRUE ) {
      /* В текущей версии FreeRTOS нет средств для корректного
       * переключения на другую сопрограмму из тела обработчика
       * прерывания! */
   }
}
/*-----------------------------------------------------------*/

/* Точка входа. С функции main() начнется выполнение про-
граммы. */
int main(void) {
   /* Создать 3 очереди для хранения элементов типа unsigned long.
    * Длина каждой очереди — 3 элемента. */
   xQueue1 = xQueueCreate(3, sizeof(unsigned long));
   xQueue2 = xQueueCreate(3, sizeof(unsigned long));
   xQueue3 = xQueueCreate(3, sizeof(unsigned long));

   /* Создать служебную сопрограмму.
    * Приоритет = 1. */
   xCoRoutineCreate(vIntCoRoutine, 1, 0);
   /* Создать сопрограммы № 1 и № 2 как экземпляры одной
сопрограммы.
    * Экземпляры различаются целочисленным параметром,
    * который передается сопрограмме при ее создании.
    * Приоритет обеих сопрограмм = 2. */
   xCoRoutineCreate(vTransferCoRoutine, 2, 1);
   xCoRoutineCreate(vTransferCoRoutine, 2, 2);

   /* Связать прерывания MS-DOS с соответствующими об-
работчиками прерываний. */
   _dos_setvect(0x82, vReceiveInterruptHandler);
   _dos_setvect(0x83, vSendInterruptHandler);

   /* Запуск планировщика. */
   vTaskStartScheduler();
   /* При нормальном выполнении программа до этого места
"не дойдет" */
   for (;;) ;
}
/*-----------------------------------------------------------*/

/* Функция, реализующая задачу Бездействие,
должна присутствовать в программе и содержать вызов
vCoRoutineSchedule() */
void vApplicationIdleHook(void) {
   /* Так как задача Бездействие не выполняет других действий,
   то вызов vCoRoutineSchedule() размещен внутри бесконеч-
ного цикла.*/
   for (;;) {
      vCoRoutineSchedule();
   }
}
```

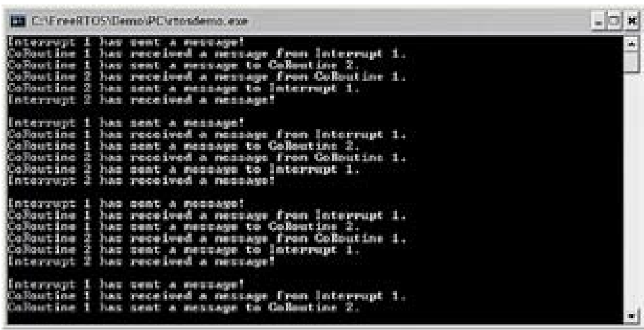

Рис. 6. Результат выполнения учебной программы № 2

По результатам работы учебной программы (рис. 6) можно проследить, как сообщение генерируется сначала в Прерывании № 1, затем передается в Сопрограмму № 1 и далее — в Сопрограмму № 2, которая в свою очередь отсылает сообщение Прерыванию № 2.

Время реакции системы на события

Продемонстрируем недостаток кооперативной многозадачности по сравнению с вытесняющей с точки зрения времени реакции системы на прерывания. Для этого заменим реализацию служебной сопрограммы в учебной программе № 2 ***vIntCoRoutine()*** на следующую:

```
/* Служебная сопрограмма. Вызывает программные прерывания.
 * Приоритет = 1.*/
void vIntCoRoutine( xCoRoutineHandle xHandle, unsigned
portBASE_TYPE uxIndex ) {
   /* Все переменные должны быть объявлены как static. */
   static long i;
   crSTART( xHandle );
   for(;;) {
      /* Эта инструкция сгенерирует Прерывание № 1. */
      __asm {int 0x83}
      /* Грубая реализация задержки на какое-то время.
       * Служебная сопрограмма при этом не блокируется! */
      for (i = 0; i < 5000000; i++);
      /* Эта инструкция сгенерирует Прерывание № 2. */
      __asm {int 0x82}
      /* Грубая реализация задержки на какое-то время.
       * Служебная сопрограмма при этом не блокируется! */
      for (i = 0; i < 5000000; i++);
   }
   crEND();
}
/*-----------------------------------------------------------*/
```

В этом случае низкоприоритетная служебная сопрограмма не вызывает блокирующих

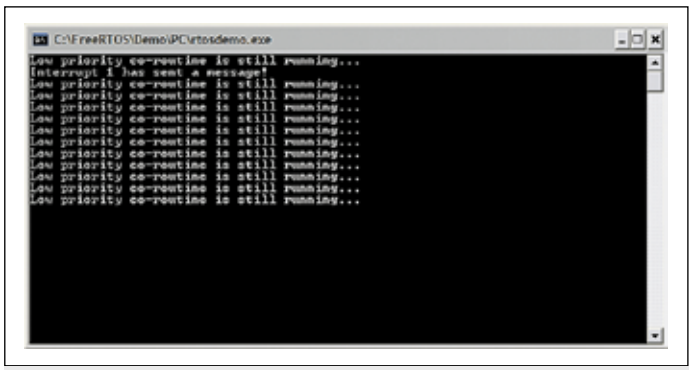

Рис. 7. Работа модифицированной учебной программы № 2

API-функций. Результат выполнения модифицированной учебной программы № 2 приведен на рис. 7.

На рис. 7 видно, что теперь выполняется только низкоприоритетная служебная сопрограмма. Высокоприоритетная Сопрограмма № 1 не получает процессорного времени, даже несмотря на то, что она вышла из блокированного состояния, когда Прерывание № 1 поместило сообщение в Очередь № 1.

Рассмотрим реальную программу, в которой высокоприоритетная сопрограмма отвечает за обработку события, ожидая, когда в очереди появится сообщение. Сообщение в очередь помещает обработчик прерывания, которое возникает при наступлении события.

Пусть в текущий момент выполняется низкоприоритетная сопрограмма и происходит это прерывание. Обработчик прерывания помещает сообщение в очередь. Однако высокоприоритетная сопрограмма не получит управления сразу же после выполнения обработчика прерывания. Высокоприоритетная сопрограмма вынуждена ожидать, пока низкоприоритетная сопрограмма отдаст управление, вызвав блокирующую API-функцию.

Таким образом, время реакции системы на событие зависит от того, насколько быстро выполняющаяся в данный момент сопрограмма выполнит переключение на другую сопрограмму. Высокоприоритетная сопрограмма вынуждена ожидать, пока выполняется низкоприоритетная.

С точки зрения времени реакции системы на внешние события кооперативная многозадачность не позволяет гарантировать заданное время реакции, что является одним из основных недостатков кооперативной многозадачности.

Выводы

Подводя итог, можно выделить следующие тезисы относительно сопрограмм во FreeRTOS:

- Выполняются в режиме кооперативной многозадачности.
- Значительно экономят оперативную память.
- Автоматически устраняют проблему реентерабельности функций.
- Не гарантируют заданного времени реакции системы на прерывание.
- При написании сопрограмм следует придерживаться строгих ограничений.
- Бедный набор API-функций для работы с сопрограммами.

Таким образом, использование сопрограмм может быть оправдано лишь в том случае, если преследуется цель написания программы, работающей под управлением FreeRTOS, на микроконтроллере, который не имеет достаточного объема оперативной памяти для реализации программы с использованием задач.

Литература

1. Курниц А. FreeRTOS — операционная система для микроконтроллеров // Компоненты и технологии. 2011. № 2–8.
2. www.freertos.org
3. http://www.ee.ic.ac.uk/t.clarke/rtos/lectures/RTOSlec2x2bw.pdf