FreeRTOS — операционная система для микроконтроллеров

Андрей КУРНИЦ
kurnits@stim.by

Это очередная статья из цикла, посвященного FreeRTOS — операционной системе для микроконтроллеров. Здесь читатель познакомится с нововведением последних версий FreeRTOS — встроенной реализацией программных таймеров.

Что представляет собой программный таймер?

В версии FreeRTOS V7.0.0 по сравнению с предыдущими версиями появилось существенное нововведение — встроенная реализация программных таймеров. Программный таймер (далее по тексту — таймер) во FreeRTOS — это инструмент, позволяющий организовать выполнение подпрограммы в точно заданные моменты времени.

Часть программы, выполнение которой инициирует таймер, в программе представлена в виде функции языка Си, которую в дальнейшем мы будем называть функцией таймера. Функция таймера является функцией обратного вызова (callback function). Механизм программных таймеров обеспечивает вызов функции таймера в нужные моменты времени.

Программные таймеры предоставляют более удобный способ привязки выполнения программы к заданным моментам времени, чем использование API-функций ***vTaskDelay()*** и ***vTaskDelayUntil()***, которые переводят задачу в блокированное состояние на заданный промежуток времени [1, № 4].

Принцип работы программного таймера

Как и прочие объекты ядра FreeRTOS, программный таймер должен быть создан до первого своего использования в программе. При создании таймера с ним связывается функция таймера, выполнение которой он будет инициировать.

Таймер может находиться в двух состояниях: пассивном (Dorman state) и активном (Active state).

Пассивное состояние таймера характеризуется тем, что таймер в данный момент не отсчитывает временной интервал. Таймер, находящийся в пассивном состоянии, никогда не вызовет свою функцию. Сразу после создания таймер находится в пассивном состоянии.

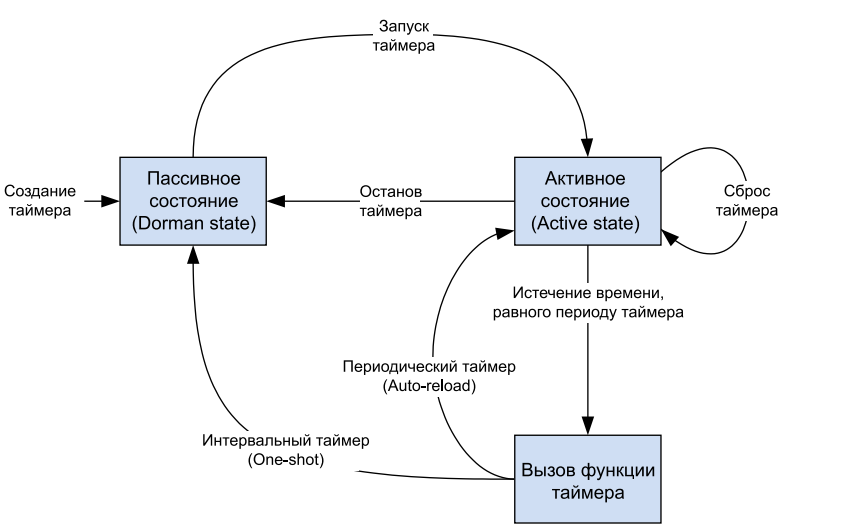

Рис. 1. Операции с таймером, состояния таймера и переходы между ними

Таймер переходит в активное состояние после того, как к нему в явном виде применили операцию запуска таймера. Таймер, находящийся в активном состоянии, рано или поздно вызовет свою функцию таймера. Промежуток времени от момента запуска таймера до момента, когда он автоматически вызовет свою функцию, называется периодом работы таймера. Период таймера задается в момент его создания, но может быть изменен в ходе выполнения программы. Момент времени, когда таймер вызывает свою функцию, будем называть моментом срабатывания таймера.

Рассматривая таймер в упрощенном виде, можно сказать, что к таймеру, находящемуся в пассивном состоянии, применяют операцию запуска, в результате которой таймер переходит из пассивного состояния в активное и начинает отсчитывать время. Когда с момента запуска таймера пройдет промежуток времени, равный периоду работы таймера, то таймер сработает и автоматически вызовет свою функцию таймера (рис. 1).

К таймеру могут быть применены следующие операции:

1. Создание таймера — приводит к выделению памяти под служебную структуру управления таймером, связывает таймер с его функцией, которая будет вызываться при срабатывании таймера, переводит таймер в пассивное состояние.
2. Запуск — переводит таймер из пассивного состояния в активное, таймер начинает отсчет времени.
3. Останов — переводит таймер из активного состояния в пассивное, таймер прекращает отсчет времени, функция таймера так и не вызывается.
4. Сброс — приводит к тому, что таймер начинает отсчет временного интервала с начала. Подробнее об этой операции расскажем позже.
5. Изменение периода работы таймера.
6. Удаление таймера — приводит к освобождению памяти, занимаемой служебной структурой управления таймером.

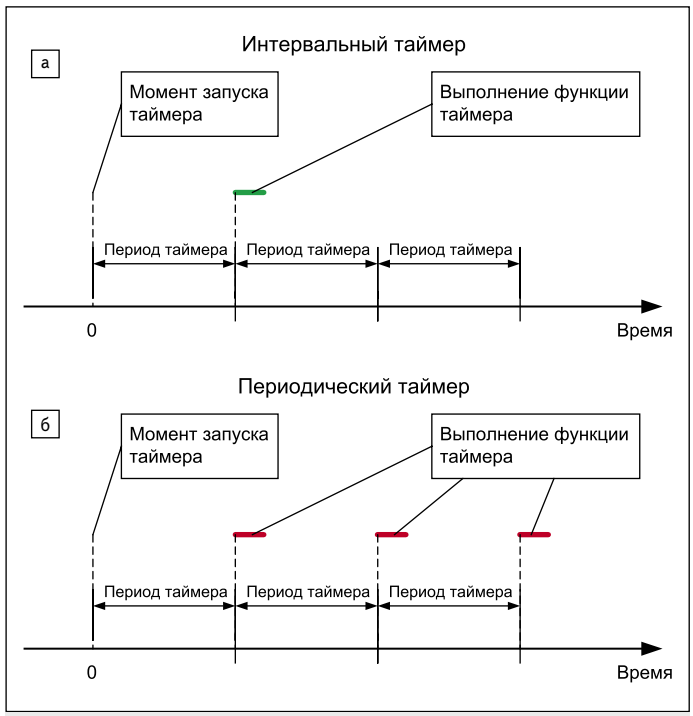

Рис. 2. Работа интервального и периодического таймера

Режимы работы таймера

Таймеры во FreeRTOS различаются по режиму работы в зависимости от состояния, в которое переходит таймер после того, как произошло его срабатывание. Программный таймер во FreeRTOS может работать в одном из двух режимов:

- режим интервального таймера (One-shot timer);
- режим периодического таймера (Auto-reload timer).

Интервальный таймер

Характеризуется тем, что после срабатывания таймера он переходит в пассивное состояние. Таким образом, функция таймера будет вызвана один раз — когда время, равное периоду таймера, истечет. Однако после этого интервальный таймер можно «вручную» запустить заново, но автоматически этого не происходит (рис. 2а).

Интервальный таймер применяют, когда необходимо организовать однократное выполнение какого-либо действия спустя заданный промежуток времени, который отсчитывается с момента запуска таймера.

Периодический таймер

Характеризуется тем, что после срабатывания таймера он остается в активном состоянии и начинает отсчет временного интервала с начала. Можно сказать, что после срабатывания периодический таймер сам автоматически запускается заново. Таким образом, единожды запущенный периодический таймер реализует циклическое выполнение функции таймера с заданным периодом (рис. 2б).

Периодический таймер применяют, когда необходимо организовать циклическое, повторяющееся выполнение определенных действий с точно заданным периодом.

Режим работы таймера задается в момент его создания и не может быть изменен в процессе выполнения программы.

Сброс таймера и изменение периода

Во FreeRTOS есть возможность сбросить таймер после того, как он уже запущен. В результате сброса таймер начнет отсчитывать временной интервал (равный периоду таймера) не с момента, когда таймер был запущен, а с момента, когда произошел его сброс (рис. 3).

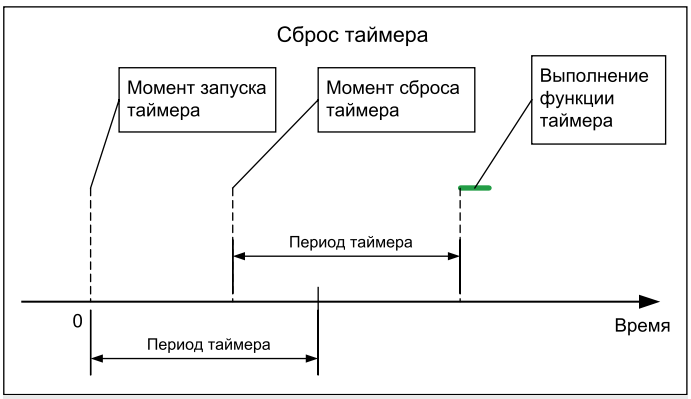

Рис. 3. Влияние сброса таймера на отсчет времени

Типичный пример использования операции сброса таймера — в устройстве, содержащем ЖКИ-дисплей с подсветкой. Подсветка дисплея включается по нажатию любой клавиши, а выключается спустя, например, 5 с после последнего нажатия. Если для отсчета 5 с использовать интервальный таймер, то операция сброса этого таймера должна выполняться при нажатии любой клавиши (подсветка в это время включена). Функция таймера должна реализовывать выключение подсветки. В этом случае, пока пользователь нажимает на клавиши, таймер сбрасывается и начинает отсчет 5 с с начала. Как только с момента последнего нажатия на клавишу прошло 5 с, выполнится функция таймера, и подсветка будет выключена.

Операция изменения периода работы таймера подобна операции сброса. При изменении периода отсчет времени также начинается с начала, отличие заключается лишь в том, что таймер начинает отсчитывать другой, новый период времени. Таким образом, время, прошедшее от момента запуска до момента изменения периода, не учитывается: новый период начинает отсчитываться с момента его изменения (рис. 4).

На рис. 4б видно, что в результате изменения периода таймер не срабатывает, если на момент изменения периода таймер отсчитал промежуток времени больше, чем новый период таймера.

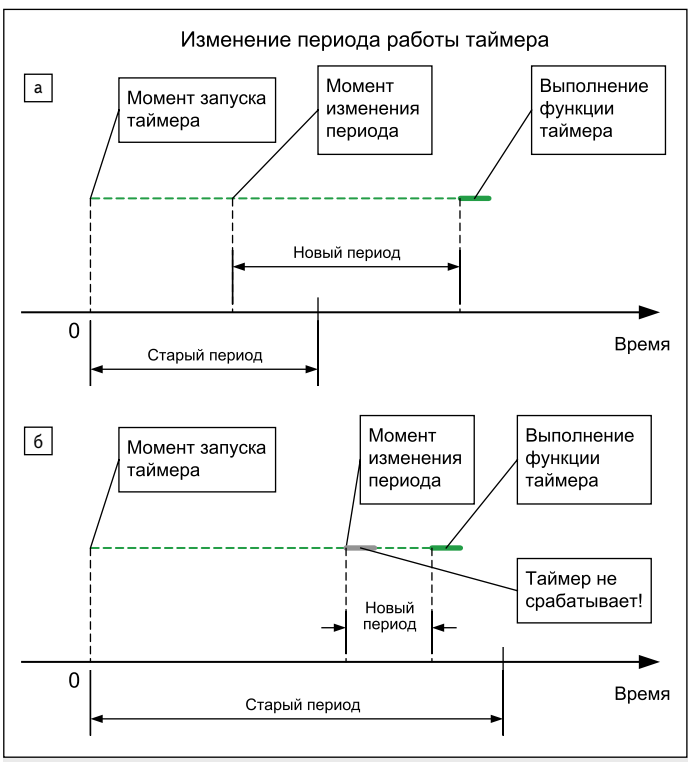

Рис. 4. Влияние изменения периода таймера на отсчет времени

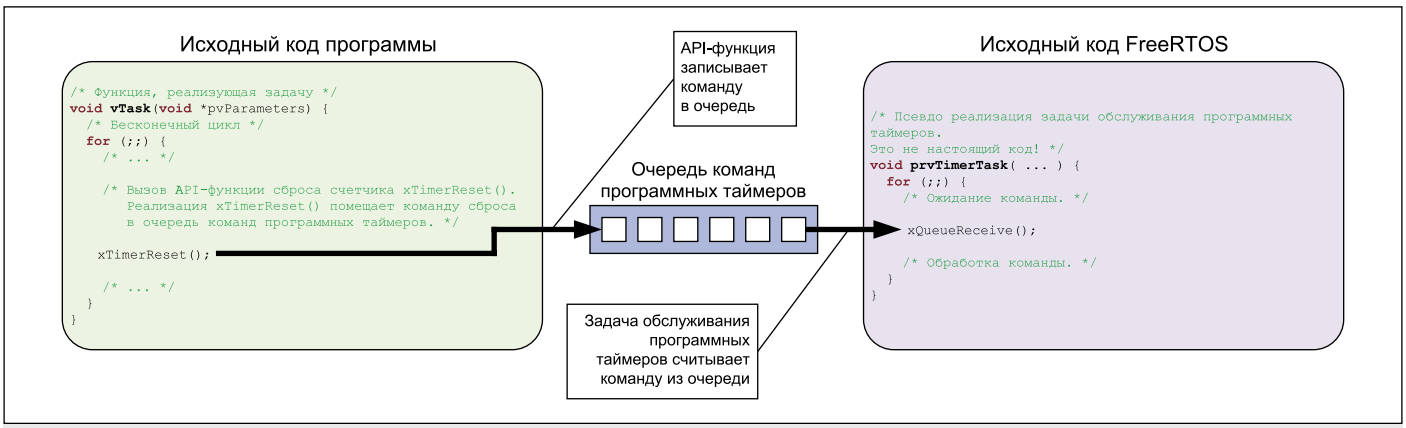

Рис. 5. Передача команды вследствие вызова API-функции сброса таймера

Реализация программных таймеров во FreeRTOS

Функция таймера

При срабатывании таймера автоматически происходит вызов функции таймера. Функция таймера реализуется в программе в виде функции языка Си, она должна иметь следующий прототип:

```
void vTimerCallbackFunction( xTimerHandle xTimer );
```

В отличие от функций, реализующих задачи и сопрограммы, функция таймера не должна содержать бесконечного цикла. Напротив, ее выполнение должно происходить как можно быстрее:

```
void vTimerCallbackFunction( xTimerHandle xTimer )
{
    // Код функции таймера
    return;
}
```

Единственный аргумент функции таймера — дескриптор таймера, срабатывание которого привело к вызову этой функции. Функция таймера является функцией обратного вызова (Callback function), это значит, что ее вызов происходит автоматически. Программа не должна содержать явные вызовы функции таймера. Дескриптор таймера автоматически копируется в аргумент функции таймера при ее вызове и может быть использован в теле функции таймера для операций с этим таймером.

Указатель на функцию таймера задан в виде макроопределения ***tmrTIMER_CALLBACK***.

Задача обслуживания программных таймеров

Немаловажно и то, что механизм программных таймеров фактически не является частью ядра FreeRTOS. Все программные таймеры в программе отсчитывают время и вызывают свои функции за счет того, что в программе выполняется одна дополнительная сервисная задача, которую в дальнейшем мы будем называть задачей обслуживания программных таймеров. Вызов функции таймера выполняет именно задача обслуживания таймеров.

Задача обслуживания таймеров недоступна программисту напрямую (нет доступа к ее дескриптору), она автоматически создается во время запуска планировщика, если настройки FreeRTOS предусматривают использование программных таймеров.

Большую часть времени задача обслуживания таймеров пребывает в блокированном состоянии, она разблокируется лишь тогда, когда будет вызвана API-функция работы с таймерами или сработал один из таймеров.

Ограничение на вызов API-функций из функции таймера

Так как функция таймера вызывается из задачи обслуживания таймеров, а переход последней в блокированное состояние и выход из него напрямую связан с отсчетом времени таймерами, то функция таймера никогда не должна пытаться заблокировать задачу обслуживания прерываний, то есть вызывать блокирующие API-функции.

Например, функция таймера никогда не должна вызывать API-функции ***vTaskDelay()*** и ***vTaskDelayUntil()***, а также API-функции доступа к очередям, семафорам и мьютексам с ненулевым временем тайм-аута.

Очередь команд таймеров

Для совершения операций запуска, останова, сброса, изменения периода и удаления таймеров во FreeRTOS предоставляется набор API-функций, которые могут вызываться из задач и обработчиков прерываний, а также из функций таймеров. Вызов этих API-функций не воздействует напрямую на задачу обслуживания таймеров. Вместо этого он приводит к записи команды в очередь, которую в дальнейшем мы будем называть очередью команд таймеров. Задача обслуживания таймеров считывает команды из очереди и выполняет их.

Таким образом, очередь команд выступает средством безопасного управления программными таймерами в многозадачной среде, где программные таймеры играют роль совместно используемого ресурса.

Очередь команд недоступна для прямого использования в программе, доступ к ней имеют только API-функции работы с таймерами. Рис. 5 поясняет процесс передачи команды от прикладной задачи к задаче обслуживания программных таймеров.

Как видно на рис. 5, прикладная программа не обращается к очереди напрямую, вместо этого она вызывает API-функцию сброса таймера, которая помещает команду сброса таймера в очередь команд программных таймеров. Задача обслуживания программных таймеров считывает эту команду из очереди и непосредственно сбрасывает таймер.

Важно, что таймер отсчитывает промежуток времени с момента, когда была вызвана соответствующая API-функция, а не с момента, когда команда была считана из очереди. Это достигается за счет того, что в очередь команд помещается информация о значении счетчика системных квантов.

Дискретность отсчета времени

Программные таймеры во FreeRTOS реализованы на основе уже имеющихся объектов ядра: на основе задачи и очереди, управление которыми осуществляет планировщик. Работа планировщика жестко привязана к системному кванту времени. Поэтому нет ничего удивительного в том, что програм-мные таймеры отсчитывают промежутки времени, кратные одному системному кванту.

То есть минимальный промежуток времени, который может быть отсчитан программным таймером, составляет один системный квант времени.

Эффективность реализации программных таймеров

Подведя промежуточный итог, можно выделить основные тезисы касательно реализации программных таймеров во FreeRTOS:

1. Для всех программных таймеров в программе используется одна-единственная задача обслуживания таймеров и одна-единственная очередь команд.

2. Функция таймера выполняется в контексте задачи обслуживания таймеров, а не в контексте обработчика прерывания микроконтроллера.
3. Процессорное время не расходуется задачей обслуживания таймеров, когда происходит отсчет времени. Задача обслуживания таймеров получает управление, лишь когда истекает время, равное периоду работы одного из таймеров.
4. Использование программных таймеров не добавляет никаких вычислений в обработчик прерывания от аппаратного таймера микроконтроллера, который используется для отсчета системных квантов времени.
5. Программные таймеры реализованы на существующих механизмах FreeRTOS, поэтому использование программных таймеров в программе повлечет минимальное увеличение размера скомпилированной программы.
6. Программные таймеры пригодны лишь для отсчета временных промежутков, кратных одному системному кванту времени.

Потребление оперативной памяти при использовании таймеров

Оперативная память, задействованная для программных таймеров, складывается из 3 составляющих:

1. Память, используемая задачей обслуживания таймеров. Ее объем не зависит от количества таймеров в программе.
2. Память, используемая очередью команд программных таймеров. Ее объем также не зависит от количества таймеров.
3. Память, выделяемая для каждого вновь создаваемого таймера. В ней размещается структура управления таймером ***xTIMER***. Объем этой составляющей пропорционален числу созданных в программе таймеров.

Рассчитаем объем памяти, который потребуется для добавления в программу 10 программных таймеров. В качестве платформы выбран порт FreeRTOS для реального режима x86 процессора, который используется в учебных программах в этом цикле статей. Настройки ядра FreeRTOS идентичны настройкам демонстрационного проекта, который входит в дистрибутив FreeRTOS.

Память, используемая задачей обслуживания таймеров, складывается из памяти, занимаемой блоком управления задачей tskTCB, — 70 байт и памяти стека, примем его равным минимальному рекомендованному ***configMINIMAL_STACK_SIZE*** = 256 слов (16-битных), что равно 512 байт. В сумме получаем 70 + 512 = 582 байт.

Память, используемая очередью команд таймеров, складывается из памяти для размещения блока управления очередью ***xQUEUE*** — 58 байт и памяти, в которой разместятся элементы очереди. Элемент очереди команд представляет собой структуру типа ***xTIMER_MESSAGE***, размер которой равен 8 байт. Пусть используется очередь длиной 10 команд, тогда для размещения их в памяти потребуется 8×10 = 80 байт. В сумме получаем 58+80 = 138 байт.

Каждый таймер в программе обслуживается с помощью структуры управления таймером ***xTIMER***, ее размер составляет 34 байт. Так как таймеров в программе 10, то памяти потребуется 34×10 = 340 байт.

Итого при условиях, оговоренных выше, для добавления в программу 10 программных таймеров потребуется 582+138+340 = = 1060 байт оперативной памяти.

Настройки FreeRTOS для использования таймеров

Чтобы использовать программные таймеры в своей программе, необходимо сделать следующие настройки FreeRTOS. Файл с исходным кодом программных таймеров ***/Source/timers.c*** должен быть включен в проект. Кроме того, в исходный текст программы должен быть включен заголовочный файл ***croutine.h***, содержащий прототипы API-функций для работы с таймерами:

```
#include “timers.h”
```

Также в файле конфигурации ***FreeRTOSConfig.h*** должны присутствовать следующие макроопределения:

1. ***configUSE_TIMERS***. Определяет, включены ли программные таймеры в конфигурацию FreeRTOS: 1 — включены, 0 — исключены. Помимо прочего определяет, будет ли автоматически создана задача обслуживания таймеров в момент запуска планировщика.
2. ***configTIMER_TASK_PRIORITY***. Задает приоритет задачи обслуживания таймеров. Как и для всех задач, приоритет задачи обслуживания таймеров может находиться в пределах от 0 до (***configMAX_PRIORITIES–1***). Значение приоритета задачи обслуживания таймеров необходимо выбирать с осторожностью, учитывая требования к создаваемой программе. Например, если задан наивысший в программе приоритет, то команды задаче обслуживания таймеров будут передаваться без задержек, а функция таймера будет вызываться сразу же, когда время, равное периоду таймера, истекло. Наоборот, если задаче обслуживания таймеров назначен низкий приоритет, то передача команд и вызов функции таймера будут задержаны по времени, если в данный момент выполняется задача с более высоким приоритетом.
3. ***configTIMER_QUEUE_LENGTH***. Размер очереди команд — устанавливает максимальное число невыполненных команд, которые могут храниться в очереди, прежде чем задача обслуживания таймеров их выполнит. Размер очереди зависит от количества вызовов API-функций для работы с таймерами во время, когда функция обслуживания таймеров не выполняется. А именно когда:
   - Планировщик еще не запущен или приостановлен.
   - Происходит несколько вызовов API-функций для работы с таймерами из обработчиков прерываний, так как когда процессор занят выполнением обработчика прерывания, ни одна задача не выполняется.
   - Происходит несколько вызовов API-функций для работы с таймерами из задачи (задач), приоритет которых выше, чем у задачи обслуживания таймеров.
4. ***configTIMER_TASK_STACK_DEPTH***. Задает размер стека задачи обслуживания таймеров. Задается не в байтах, а в словах, равных разрядности процессора. Тип данных слова, которое хранится в стеке, задан в виде макроопределения ***portSTACK_TYPE*** в файле ***portmacro.h***. Функция таймера выполняется в контексте задачи обслуживания таймеров, поэтому размер стека задачи обслуживания таймеров определяется потреблением памяти стека функциями таймеров.

Работа с таймерами

Как и для объектов ядра, таких как задачи, сопрограммы, очереди и др., для работы с программным таймером служит дескриптор (handle) таймера.

Дескриптор таймера представляет собой переменную типа ***xTimerHandle***. При создании таймера FreeRTOS автоматически назначает ему дескриптор, который далее используется в программе для операций с этим таймером.

Функция таймера автоматически получает дескриптор таймера в качестве своего аргумента. Для выполнения операций с таймером внутри функции этого таймера следует использовать дескриптор таймера, полученный в виде аргумента.

Дескриптор таймера однозначно определяет таймер в программе. Тем не менее при создании таймера ему можно назначить идентификатор. Идентификатор представляет собой указатель типа ***void****, что подразумевает использование его как указателя на любой тип данных. Идентификатор таймера следует использовать лишь тогда, когда необходимо связать таймер с произвольным параметром. Например, можно создать несколько таймеров с общей для них всех функцией таймера, а идентификатор использовать внутри функции таймера для определения того, срабатывание какого конкретно таймера привело к вызову этой функции. Такое использование идентификатора будет продемонстрировано ниже в учебной программе.

Создание/удаление таймера

Для того чтобы создать программный таймер, следует вызвать API-функцию ***xTimerCreate()***. Ее прототип:

```
xTimerHandle xTimerCreate( const signed char *pcTimerName, portTickType xTimerPeriod, unsigned portBASE_TYPE uxAutoReload, void * pvTimerID, tmrTIMER_CALLBACK pxCallbackFunction );
```

Аргументы и возвращаемое значение:

1. ***pcTimerName*** — нультерминальная (заканчивающаяся нулем) строка, определяющая имя таймера. Ядром не используется, а служит лишь для наглядности и при отладке.
2. ***xTimerPeriod*** — период работы таймера. Задается в системных квантах времени, для задания в миллисекундах следует использовать макроопределение ***portTICK_RATE_MS***. Например, для задания периода работы таймера равным 500 мс следует присвоить аргументу ***xTimerPeriod*** значение выражения ***500/portTICK_RATE_MS***. Нулевое значение периода работы таймера не допускается.
3. ***uxAutoReload*** — определяет тип создаваемого таймера. Может принимать следующие значения:
   - ***pdTRUE*** — будет создан периодический таймер.
   - ***pdFALSE*** — будет создан интервальный таймер.
4. ***pvTimerID*** — задает указатель на идентификатор, который будет присвоен создаваемому экземпляру таймера. Этот аргумент следует использовать при создании нескольких экземпляров таймеров, которым соответствует одна-единственная функция таймера.
5. ***pxCallbackFunction*** — указатель на функцию таймера, фактически — имя функции в программе. Функция таймера должна иметь следующий прототип:

```
void vCallbackFunction( xTimerHandle xTimer );
```

Указатель на функцию таймера задан также в виде макроопределения ***tmrTIMER_CALLBACK***.

6. Возвращаемое значение. Если таймер успешно создан, возвращаемым значением будет ненулевой дескриптор таймера. Если же таймер не создан по причине нехватки оперативной памяти или при задании периода таймера равным нулю, то возвращаемым значением будет 0.

Важно, что таймер после создания находится в пассивном состоянии. API-функция ***xTimerCreate()*** действует непосредственно и не использует очередь команд таймеров.

Ранее созданный таймер может быть удален. Для этого предназначена API-функция ***xTimerDelete()***. Ее прототип:

```
portBASE_TYPE xTimerDelete( xTimerHandle xTimer, portTickType xBlockTime );
```

Аргументы и возвращаемое значение:

1. ***xTimer*** — дескриптор таймера, полученный при его создании API-функцией ***xTimerCreate()***.
2. ***xBlockTime*** — определяет время тайм-аута — максимальное время нахождения вызывающей ***xTimerDelete()*** задачи в блокированном состоянии, если очередь команд полностью заполнена и нет возможности поместить в нее команду об уничтожении таймера.
3. Возвращаемое значение — может принимать два значения:

• ***pdFAIL*** — означает, что команда об удалении так и не была помещена в очередь команд, а время тайм-аута истекло.

• ***pdPASS*** — означает, что команда об удалении успешно помещена в очередь команд.

Вызов ***xTimerDelete()*** приводит к освобождению памяти, занимаемой структурой управления таймером xTIMER.

API-функции ***xTimerCreate()*** и ***xTimerDelete()*** недопустимо вызывать из обработчиков прерываний.

Запуск/останов таймера

Запуск таймера осуществляется с помощью API-функции ***xTimerStart()***. Ее прототип:

```
portBASE_TYPE xTimerStart( xTimerHandle xTimer, portTickType xBlockTime );
```

Аргументы и возвращаемое значение:

1. ***xTimer*** — дескриптор таймера, полученный при его создании API-функцией ***xTimerCreate()***.
2. ***xBlockTime*** — определяет время тайм-аута — максимальное время нахождения вызывающей ***xTimerStart()*** задачи в блокированном состоянии, если очередь команд полностью заполнена и нет возможности поместить в нее команду о запуске таймера.
3. Возвращаемое значение — может принимать два значения:
   - ***pdFAIL*** — означает, что команда о запуске таймера так и не была помещена в очередь команд, а время тайм-аута истекло.
   - ***pdPASS*** — означает, что команда о запуске успешно помещена в очередь команд.

Запуск таймера может быть произведен и с помощью вызова API-функции ***xTimerReset()***, подробно об этом — в описании API-функции ***xTimerReset()*** ниже.

Таймер, который уже отсчитывает время, находясь в активном состоянии, может быть принудительно остановлен. Для этого предназначена API-функция ***xTimerStop()***. Ее прототип:

```
portBASE_TYPE xTimerStop( xTimerHandle xTimer, portTickType xBlockTime );
```

Аргументы и возвращаемое значение:

1. ***xTimer*** — дескриптор таймера, полученный при его создании API-функцией ***xTimerCreate()***.
2. ***xBlockTime*** — определяет время тайм-аута — максимальное время нахождения вызывающей ***xTimerStop()*** задачи в блокированном состоянии, если очередь команд полностью заполнена и нет возможности поместить в нее команду об останове таймера.
3. Возвращаемое значение — может принимать два значения:
   - ***pdFAIL*** — означает, что команда об останове таймера так и не была помещена в очередь команд, а время тайм-аута истекло.
   - ***pdPASS*** — означает, что команда об останове успешно помещена в очередь команд.

API-функции ***xTimerStart()*** и ***xTimerStop()*** предназначены для вызова из задачи или функции таймера. Существуют версии этих API-функций, предназначенные для вызова из обработчиков прерываний, о них будет сказано ниже.

Сброс таймера

Сброс таймера осуществляется с помощью API-функции ***xTimerReset()***. Ее прототип:

```
portBASE_TYPE xTimerReset( xTimerHandle xTimer, portTickType xBlockTime );
```

Аргументы и возвращаемое значение:

1. ***xTimer*** — дескриптор таймера, полученный при его создании API-функцией ***xTimerCreate()***.
2. ***xBlockTime*** — определяет время тайм-аута — максимальное время нахождения вызывающей ***xTimerReset()*** задачи в блокированном состоянии, если очередь команд полностью заполнена и нет возможности поместить в нее команду о сбросе таймера.
3. Возвращаемое значение — может принимать два значения:
   - ***pdFAIL*** — означает, что команда о сбросе таймера так и не была помещена в очередь команд, а время тайм-аута истекло.
   - ***pdPASS*** — означает, что команда о сбросе таймера успешно помещена в очередь команд.

Операция сброса может применяться как к активному таймеру, так и к находящемуся в пассивном состоянии. В случае если таймер находился в пассивном состоянии, вызов ***xTimerReset()*** будет эквивалентен вызову ***xTimerStart()***, то есть таймер будет запущен. Если таймер уже отсчитывал время в момент вызова ***xTimerReset()*** (то есть находился в активном состоянии), то вызов ***xTimerReset()*** приведет к тому, что таймер заново начнет отсчет времени с момента вызова ***xTimerReset()***.

Допускается вызов ***xTimerReset()***, когда таймер уже создан, но планировщик еще не запущен. В этом случае отсчет времени начнется не с момента вызова ***xTimerReset()***, а с момента запуска планировщика.

Легко заметить, что API-функции ***xTimerReset()*** и ***xTimerStart()*** полностью экви-

валентны. Две различные API-функции введены скорее для наглядности. Предполагается, что API-функцию ***xTimerStart()*** следует применять к таймеру в пассивном состоянии, ***xTimerReset()*** — к таймеру в активном состоянии. Однако это требование совершенно необязательно, так как обе эти функции приводят к записи одной и той же команды в очередь команд таймеров.

API-функция ***xTimerReset()*** предназначена для вызова из тела задачи или функции таймера. Существует версия этой API-функции, предназначенная для вызова из обработчика прерывания, о ней будет сказано ниже.

Изменение периода работы таймера

Независимо от того, в каком состоянии в данный момент находится таймер: в активном или в пассивном, период его работы можно изменить посредством API-функции ***xTimerChangePeriod()***. Ее прототип:

```
portBASE_TYPE xTimerChangePeriod( xTimerHandle xTimer,
portTickType xNewPeriod, portTickType xBlockTime );
```

Аргументы и возвращаемое значение:

1. ***xTimer*** — дескриптор таймера, полученный при его создании API-функцией ***xTimerCreate()***.
2. ***xNewPeriod*** — новый период работы таймера, задается в системных квантах.
3. ***xBlockTime*** — определяет время тайм-аута — максимальное время нахождения вызывающей ***xTimerChangePeriod()*** задачи в блокированном состоянии, если очередь команд полностью заполнена и нет возможности поместить в нее команду об изменении периода таймера.
4. Возвращаемое значение — может принимать два значения:
   - ***pdFAIL*** — означает, что команда об изменении периода таймера так и не была помещена в очередь команд, и время тайм-аута истекло.
   - ***pdPASS*** — означает, что команда об изменении периода успешно помещена в очередь команд.

API-функция ***xTimerChangePeriod()*** предназначена для вызова из тела задачи или функции таймера. Существует версия этой API-функции, предназначенная для вызова из обработчика прерывания, о ней будет сказано ниже.

Получение текущего состояния таймера

Для того чтобы узнать, в каком состоянии — в активном или в пассивном — в данный момент находится таймер, служит API-функция ***xTimerIsTimerActive()***. Ее прототип:

```
portBASE_TYPE xTimerIsTimerActive( xTimerHandle xTimer );
```

Аргументом API-функции является дескриптор таймера, состояние которого необходимо выяснить. ***xTimerIsTimerActive()*** может возвращать два значения:

- ***pdTRUE***, если таймер находится в активном состоянии.
- ***pdFALSE***, если таймер находится в пассивном состоянии.

API-функция ***xTimerIsTimerActive()*** предназначена для вызова только из тела задачи или функции таймера.

Получение идентификатора таймера

При создании таймеру присваивается идентификатор в виде указателя ***void****, что позволяет связать таймер с произвольной структурой данных.

API-функцию ***pvTimerGetTimerID()*** можно вызывать из тела функции таймера для получения идентификатора, в результате срабатывания которого была вызвана эта функция таймера. Прототип API-функции ***pvTimerGetTimerID()***:

```
void *pvTimerGetTimerID( xTimerHandle xTimer );
```

Аргументом является дескриптор таймера, идентификатор которого необходимо получить. ***pvTimerGetTimerID()*** возвращает указатель на сам идентификатор.

Работа с таймерами из обработчиков прерываний

Есть возможность выполнять управление таймерами из обработчиков прерываний микроконтроллера. Для рассмотренных выше API-функций ***xTimerStart()***, ***xTimerStop()***, ***xTimerChangePeriod()*** и ***xTimerReset()*** существуют версии, предназначенные для вызова из обработчиков прерываний: ***xTimerStartFromISR()***, ***xTimerStopFromISR()***, ***xTimerChangePeriodFromISR()*** и ***xTimerResetFromISR()***. Их прототипы:

```
portBASE_TYPE xTimerStartFromISR( xTimerHandle xTimer,
portBASE_TYPE *pxHigherPriorityTaskWoken );
portBASE_TYPE xTimerStopFromISR( xTimerHandle xTimer,
portBASE_TYPE *pxHigherPriorityTaskWoken );
portBASE_TYPE xTimerChangePeriodFromISR( xTimerHandle
xTimer, portTickType xNewPeriod, portBASE_TYPE
*pxHigherPriorityTaskWoken );
portBASE_TYPE xTimerResetFromISR( xTimerHandle xTimer,
portBASE_TYPE *pxHigherPriorityTaskWoken );
```

По сравнению с API-функциями, предназначенными для вызова из задач, в версиях API-функций, предназначенных для вызова из обработчиков прерываний, произошли следующие изменения в их аргументах:

1. Аргумент, который задавал время тайм-аута, теперь отсутствует, что объясняется тем, что обработчик прерывания — не задача и не может быть заблокирован на какое-то время.
2. Появился дополнительный аргумент ***pxHigherPriorityTaskWoken***. API-функции устанавливают значение ***pxHigherPriorityTaskWoken*** в ***pdTRUE***, если в данный момент выполняется задача с приоритетом меньше, чем у задачи обслуживания программных таймеров, и в результате вызова API-функции в очередь команд программных таймеров была помещена команда, вследствие чего задача обслуживания таймеров разблокировалась. В обработчике прерывания после вызова одной из вышеперечисленных API-функций необходимо отслеживать значение ***pxHigherPriorityTaskWoken***, и если оно изменилось на ***pdTRUE***, то необходимо выполнить принудительное переключение контекста задачи. Вследствие чего управление сразу же получит более высокоприоритетная задача обслуживания таймеров.

Учебная программа

Продемонстрировать использование программных таймеров позволяет следующая учебная программа, в которой происходит создание, запуск, изменение периода, а также удаление таймера.

В программе будет создан периодический таймер с периодом работы 1 с. Функция этого таймера каждый раз при его срабатывании будет увеличивать период работы на 1 секунду. Кроме того, в программе будут созданы 3 интервальных таймера с периодом работы 12 секунд каждый.

Сразу после запуска планировщика отсчет времени начнут периодический таймер и первый интервальный таймер. Через 12 с, когда сработает первый интервальный таймер, его функция запустит второй интервальный таймер, еще через 12 с функция второго интервального таймера запустит третий. Функция третьего же интервального таймера еще через 12 с удалит периодический таймер.

Таким образом, отсчет времени таймерами будет продолжаться 36 с. В моменты вызова функций таймеров на дисплей будет выводиться время, прошедшее с момента запуска планировщика.

Исходный текст учебной программы:

```
#include <stdlib.h>
#include <conio.h>
#include "FreeRTOS.h"
#include "task.h"
#include "timers.h"
/*-----------------------------------------------------------*/

/* Количество интервальных таймеров */
#define NUMBER_OF_TIMERS   3
/* Целочисленные идентификаторы интервальных таймеров */
#define ID_TIMER_1         111
#define ID_TIMER_2         222
#define ID_TIMER_3         333
/*-----------------------------------------------------------*/

/* Дескриптор периодического таймера */
xTimerHandle       xAutoReloadTimer;
/* Массив дескрипторов интервальных таймеров */
xTimerHandle       xOneShotTimers[NUMBER_OF_TIMERS];
/* Массив идентификаторов интервальных таймеров */
const unsigned portBASE_TYPE uxOneShotTimersIDs
[NUMBER_OF_TIMERS] = { ID_TIMER_1, ID_TIMER_2,
ID_TIMER_3 };
/* Период работы периодического таймера = 1 секунда */
unsigned int uiAutoReloadTimerPeriod = 1000 / portTICK_RATE_MS;
/*-----------------------------------------------------------*/

/* Функция периодического таймера.
 * Является функцией обратного вызова.
 * В программе не должно быть ее явных вызовов.
```

```
 * В функцию автоматически передается дескриптор таймера в виде аргумента xTimer. */
void vAutoReloadTimerFunction(xTimerHandle xTimer) {
   /* Сигнализировать о выполнении.
    * Вывести сообщение о текущем времени, прошедшем с момента запуска планировщика. */
   printf("AutoReload timer. Time = %d sec\n\r", xTaskGetTickCount() / configTICK_RATE_HZ);
   /* Увеличить период работы периодического таймера на 1 секунду */
   uiAutoReloadTimerPeriod += 1000 / portTICK_RATE_MS;
   /* Установить новый период работы периодического таймера.
    * Время тайм-аута (3-й аргумент) обязательно должно быть 0!
    * Так как внутри функции таймера нельзя вызывать блокирующие API-функции. */
   xTimerChangePeriod(xTimer, uiAutoReloadTimerPeriod, 0);
}
/*-----------------------------------------------------------*/

/* Функция интервальных таймеров.
 * Нескольким экземплярам интервальных таймеров соответствует одна-единственная функция.
 * Эта функция автоматически вызывается при истечении времени любого из связанных с ней таймеров.
 * Для того чтобы выяснить, время какого таймера истекло, используется идентификатор таймера. */
void vOneShotTimersFunction(xTimerHandle xTimer) {
   /* Указатель на идентификатор таймера */
   unsigned portBASE_TYPE *pxTimerID;

   /* Получить идентификатор таймера, который вызывал эту функцию таймера */
   pxTimerID = pvTimerGetTimerID(xTimer);

   /* Различные действия в зависимости от того, какой таймер вызывал функцию */
   switch (*pxTimerID) {
   /* Сработал интервальный таймер 1 */
   case ID_TIMER_1:
      /* Индикация работы + текущее время */
      printf("\t\t\t\tOneShot timer ID = %d. Time = %d sec\n\r", *pxTimerID, xTaskGetTickCount() / configTICK_RATE_HZ);
      /* Запустить интервальный таймер 2 */
      xTimerStart(xOneShotTimers[1], 0);
      break;
   /* Сработал интервальный таймер 2 */
   case ID_TIMER_2:
      /* Индикация работы + текущее время */
      printf("\t\t\t\tOneShot timer ID = %d. Time = %d sec\n\r", *pxTimerID, xTaskGetTickCount() / configTICK_RATE_HZ);
      /* Запустить интервальный таймер 3 */
      xTimerStart(xOneShotTimers[2], 0);
      break;
   case ID_TIMER_3:
      /* Индикация работы + текущее время */
      printf("\t\t\t\tOneShot timer ID = %d. Time = %d sec\n\r", *pxTimerID,
         xTaskGetTickCount() / configTICK_RATE_HZ);
      puts("\n\r\t\t\t\tAbout to delete AutoReload timer!");
      fflush();
      /* Удалить периодический таймер.
       * После этого активных таймеров в программе не останется. */
      xTimerDelete(xAutoReloadTimer, 0);
      break;
   }
}
/*-----------------------------------------------------------*/

/* Точка входа в программу. */
short main( void )
{
   unsigned portBASE_TYPE i;

   /* Создать периодический таймер.
    * Период работы таймера = 1 секунда.
    * Идентификатор таймера не используется (0). */
   xAutoReloadTimer = xTimerCreate("AutoReloadTimer", uiAutoReloadTimerPeriod, pdTRUE, 0,
vAutoReloadTimerFunction);
   /* Выполнить сброс периодического таймера ДО запуска планировщика.
    * Таким образом, он начнет отсчет времени одновременно с запуском планировщика. */
   xTimerReset(xAutoReloadTimer, 0);

   /* Создать 3 экземпляра интервальных таймеров.
    * Период работы таймеров = 12 секунд.
    * Каждому из них передать свой идентификатор.
    * Функция для них всех одна — vOneShotTimersFunction(). */
   for (i = 0; i < NUMBER_OF_TIMERS; i++) {
      xOneShotTimers[i] = xTimerCreate("OneShotTimer_n", 12000 / portTICK_RATE_MS, pdFALSE,
(void*) &uxOneShotTimersIDs[i], vOneShotTimersFunction);
   }
```

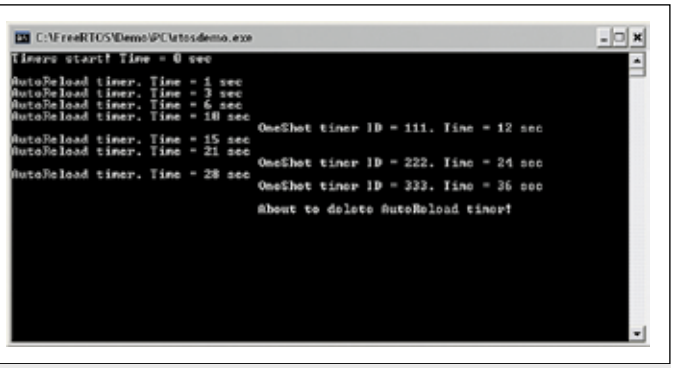

Рис. 6. Выполнение учебной программы

```
   /* Выполнить сброс только первого интервального таймера.
    * Именно он начнет отсчитывать время сразу после запуска планировщика.
    * Остальные 2 таймера после запуска планировщика останутся в пассивном состоянии. */
   xTimerReset(xOneShotTimers[0], 0);

   /* Индицировать текущее время.
    * Оно будет равно 0, так как планировщик еще не запущен. */
   printf("Timers start! Time = %d sec\n\r\n\r", xTaskGetTickCount() / configTICK_RATE_HZ);

   /* Запуск планировщика.
    * Автоматически будет создана задача обслуживания таймеров.
    * Таймеры, которые были переведены в активное состояние (например, вызовом xTimerReset())
    * ДО этого момента, начнут отсчет времени. */
   vTaskStartScheduler();

   return 1;
}
/*-----------------------------------------------------------*/
```

Для корректной компиляции учебной программы конфигурационный файл ***FreeRTOSConfig.h*** должен содержать следующие строки:

```
#define configUSE_TIMERS                1
#define configTIMER_TASK_PRIORITY       1
#define configTIMER_QUEUE_LENGTH        ( 10 )
#define configTIMER_TASK_STACK_DEPTH    configMINIMAL_STACK_SIZE
```

Результат работы учебной программы приведен на рис. 6.

В учебной программе демонстрируется прием, когда запуск (в данном случае сброс, как было сказано выше — не имеет значения) таймеров производится ДО запуска планировщика. В этом случае таймеры начинают отсчет времени сразу после старта планировщика.

В графическом виде работа учебной программы представлена на рис. 7.

Учебная программа демонстрирует также разницу между интервальными и периодическими таймерами. Как видно на рис. 6 и 7, будучи единожды запущен, интервальный таймер вызовет свою функцию один раз. Периодический же таймер напротив — вызывает свою функцию до тех пор, пока не будет удален или остановлен.

Справедливости ради следует отметить, что на практике можно обойтись без использования идентификатора таймера для определения, какой таймер вызвал функцию таймера, как это сделано в учебной программе.

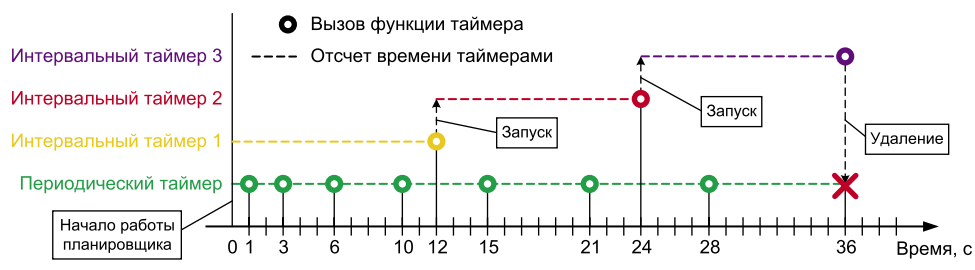

Рис. 7. Отсчет временных промежутков в учебной программе

Для этой цели вполне достаточно использовать дескриптор таймера. В таком случае функция интервальных таймеров в модифицированной учебной программе примет вид:

```
/* Функция интервальных таймеров.
 * Нескольким экземплярам интервальных таймеров соответствует одна-единственная функция.
 * Эта функция автоматически вызывается при истечении времени любого из связанных с ней таймеров.
 * Для того чтобы выяснить, время какого таймера истекло, используется идентификатор таймера. */
void vOneShotTimersFunction(xTimerHandle xTimer) {
   /* Различные действия в зависимости от того, какой таймер вызывал функцию */
   /* Сработал интервальный таймер 1? */
   if (xTimer == xOneShotTimers[0]) {
      /* Индикация работы + текущее время */
      printf("\t\t\t\tOneShot timer ID = %d. Time = %d sec\n\r", *pxTimerID, xTaskGetTickCount() /
configTICK_RATE_HZ);
      xTimerChangePeriod(xAutoReloadTimer, 6000, 0);

      /* Запустить интервальный таймер 2 */
      xTimerStart(xOneShotTimers[1], 0);
   /* Сработал интервальный таймер 2? */
   } else if (xTimer == xOneShotTimers[1]) {
      /* Индикация работы + текущее время */
      printf("\t\t\t\tOneShot timer ID = %d. Time = %d sec\n\r", *pxTimerID, xTaskGetTickCount() /
configTICK_RATE_HZ);
      /* Запустить интервальный таймер 3 */
      xTimerStart(xOneShotTimers[2], 0);
   /* Сработал интервальный таймер 3? */
   } else if (xTimer == xOneShotTimers[2]) {
      /* Индикация работы + текущее время */
      printf("\t\t\t\tOneShot timer ID = %d. Time = %d sec\n\r", *pxTimerID, xTaskGetTickCount() /
configTICK_RATE_HZ);
      puts("\n\r\t\t\t\tAbout to delete AutoReload timer!");
      fflush();
      /* Удалить периодический таймер.
      * После этого активных таймеров в программе не останется. */
      xTimerDelete(xAutoReloadTimer, 0);
   }
}
```

Дескрипторы созданных интервальных таймеров хранятся в глобальном массиве. Кроме того, дескриптор таймера, который привел к вызову функции таймера, передается в эту функцию в виде ее аргумента. Поэтому выполняя сравнение аргумента функции таймера с дескриптором, который хранится в глобальной переменной, можно сделать вывод о том, какой конкретно таймер инициировал вызов этой функции таймера.

Результат выполнения модифицированной учебной программы ничем не будет отличаться от приведенного на рис. 6, что подтверждает, что для однозначной идентификации таймера вполне достаточно иметь его дескриптор.

Выводы

Подводя итог можно сказать, что их применение оправдано в случаях, когда к точности отмеряемых временных интервалов не предъявляется высоких требований, так как активность других задач в программе может существенно повлиять на точность работы программных таймеров. Кроме того, немаловажным ограничением является дискретность работы таймеров величиной в один системный квант времени.

В дальнейших публикациях речь пойдет о способах отладки программы, которая выполняется под управлением FreeRTOS. Внимание будет сконцентрировано на:

- способах трассировки программы;
- получении статистики выполнения в реальном времени;
- способах измерения потребляемого задачей объема стека и способах защиты от его переполнения.

Литература

1. Курниц А. FreeRTOS — операционная система для микроконтроллеров // Компоненты и технологии. 2011. № 2–9.
2. www.freertos.org
3. http://ru.wikipedia.org/wiki/FreeRTOS
4. http://electronix.ru/forum/index.php?showforum=189
5. http://sourceforge.net/projects/freertos/files/FreeRTOS/
6. http://www.ee.ic.ac.uk/t.clarke/rtos/lectures/RTOSlec2x2bw.pdf