Продолжение. Начало в № 2`2011

FreeRTOS — операционная система для микроконтроллеров

В предыдущей части статьи [1] читатель познакомился с операционной системой реального времени (ОСРВ) для микроконтроллеров (МК) FreeRTOS. Были изложены достоинства и недостатки использования ОСРВ в основе программного обеспечения микроконтроллерных устройств. Произведена оценка FreeRTOS с точки зрения потребления памяти и дополнительных затрат процессорного времени. В сокращенном виде была описана структура дистрибутива FreeRTOS и назначение отдельных файлов, входящих в дистрибутив. Во второй части статьи будут затронуты как основы теории работы ОСРВ в целом, так и продолжено изучение конкретной реализации ОСРВ для МК — FreeRTOS. Уделено особое внимание задачам как базовой единице программы для FreeRTOS. Приведен пример простейшей программы для МК AVR ATmega128, работающей под управлением FreeRTOS.

Андрей КУРНИЦ
kurnits@stim.by

Основы работы ОСРВ

Прежде чем говорить об особенностях FreeRTOS, следует остановиться на основных принципах работы любой ОСРВ и пояснить значение терминов, которые будут применяться в дальнейшем. Эта часть статьи будет особенно полезна читателям, которые не знакомы с принципами, заложенными в ОСРВ.

Основой ОСРВ является ядро (Kernel) операционной системы. Ядро реализует основополагающие функции любой ОС. В ОС общего назначения, таких как Windows и Linux, ядро позволяет нескольким пользователям выполнять множество программ на одном компьютере одновременно.

Каждая выполняющаяся программа представляет собой задачу (Task). Если ОС позволяет одновременно выполнять множество задач, она является мультизадачной (Multitasking).

Большинство процессоров могут выполнять только одну задачу в один момент времени. Однако при помощи быстрого переключения между задачами достигается эффект параллельного выполнения всех задач. На рис. 1 показано истинно параллельное выполнение трех задач. В реальном же процессоре при работе ОСРВ выполнение задач носит периодический характер: каждая задача выполняется определенное время, после чего процессор «переключается» на следующую задачу (рис. 2).

Планировщик (Scheduler) — это часть ядра ОСРВ, которая определяет, какая из задач, готовых к выполнению, выполняется в данный конкретный момент времени. Планировщик может приостанавливать, а затем снова возобновлять выполнение задачи в течение всего ее жизненного цикла (то есть с момента создания задачи до момента ее уничтожения).

Алгоритм работы планировщика (Scheduling policy) — это алгоритм, по которому функционирует планировщик для принятия решения, какую задачу выполнять в данный момент времени. Алгоритм работы планировщика в ОС общего назначения заключается в предоставлении каждой задаче процессорного времени в равной пропорции. Алгоритм работы планировщика в ОСРВ отличается и будет описан ниже.

Среди всех задач в системе в один момент времени может выполняться только одна задача. Говорят, что она находится в состоянии выполнения. Остальные задачи в этот момент не выполняются, ожидая, когда планировщик выделит каждой из них процессорное время. Таким образом, задача может находиться в двух основных состояниях: выполняться и не выполняться.

Кроме того, что выполнение задачи может быть приостановлено планировщиком принудительно, задача может сама приостановить свое выполнение. Это происходит в двух случаях. Первый — это когда задача «хочет» задержать свое выполнение на определенный промежуток времени (в таком случае она переходит в состояние сна (sleep)). Второй — когда задача ожидает освобождения какого-либо аппаратного ресурса (например, последовательного порта) или наступления какого-то события (event), в этом случае говорят, что задача блокирована (block). Блокированная или «спящая» задача не нуждается в процессорном времени до наступления соответствующего события или истечения определенного интервала времени. Функции измерения интервалов времени и обслуживания событий берет на себя ядро ОСРВ.

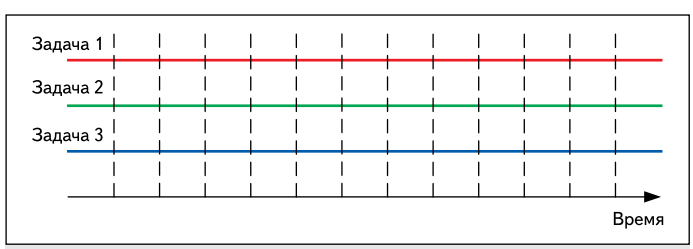

Рис. 1. Истинно параллельное выполнение задач

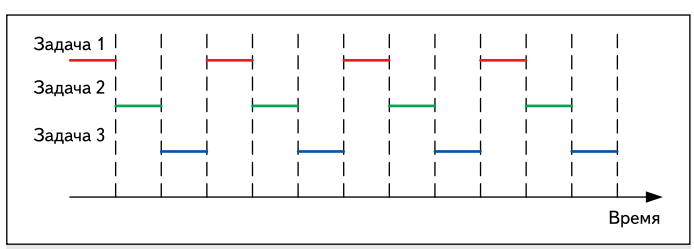

Рис. 2. Распределение процессорного времени между несколькими задачами в ОСРВ

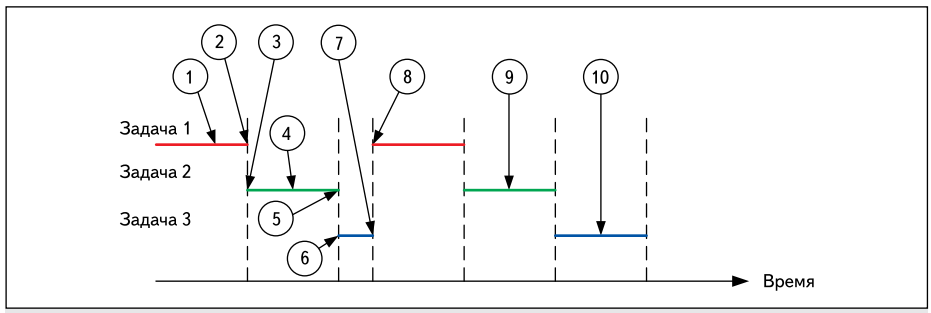

Рис. 3. Переключение между задачами, которые используют один и тот же аппаратный ресурс

Пример перехода задачи в блокированное состояние показан на рис. 3.

Задача 1 исполняется на протяжении определенного времени (1). В момент времени (2) планировщик приостанавливает задачу 1 и возобновляет выполнение задачи 2 (момент времени (3)). Во время своего выполнения (4) задача 2 захватывает определенный аппаратный ресурс для своего единоличного использования. В момент времени (5) планировщик приостанавливает задачу 2 и восстанавливает задачу 3 (6). Задача 3 пытается получить доступ к тому же самому аппаратному ресурсу, который занят задачей 2. В результате чего задача 3 блокируется — момент времени (7). Через некоторое время управление снова получает задача 2, которая завершает работу с аппаратным ресурсом и освобождает его (9). Когда управление получает задача 3, она обнаруживает, что аппаратный ресурс свободен, захватывает его и выполняется до того момента, пока не будет приостановлена планировщиком (10).

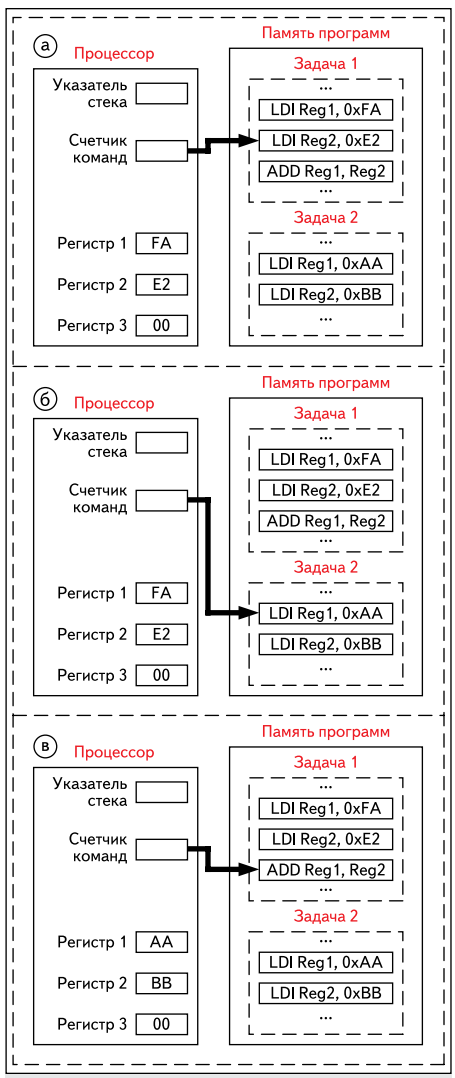

Рис. 4. Переключение между задачами без переключения контекста

Когда задача выполняется, она, как и любая программа, использует регистры процессора, память программ и память данных. Вместе эти ресурсы (регистры, стек и др.) образуют контекст задачи (task execution context). Контекст задачи целиком и полностью описывает текущее состояние процессора: флаги процессора, какая инструкция сейчас выполняется, какие значения загружены в регистры процессора, где в памяти находится вершина стека и т. д.

Задача «не знает», когда ядро ОСРВ приостановит ее выполнение или, наоборот, возобновит.

На рис. 4а показан абстрактный процессор, который выполняет задачу 1, частью которой является операция сложения. Операнды загружены в регистры Reg1 и Reg2 (инструкции LDI). Пусть перед инструкцией сложения ADD ядро приостановило задачу 1 и отдало управление задаче 2, которая использует регистры Reg1 и Reg2 для своих нужд (рис. 4б). В какой-то момент времени ядро возобновит выполнение задачи 1 с места, где она была приостановлена: с инструкции ADD (рис. 4в). Однако для задачи 1 изменение ее контекста (регистров Reg1 и Reg2) останется незамеченным, произойдет сложение, но его результат «с точки зрения» задачи 1 окажется неверным.

Таким образом, одна из основных функций ядра ОСРВ — это обеспечение идентичности контекста задачи до ее приостановки и после ее восстановления. Когда ядро приостанавливает задачу, оно должно сохранить контекст задачи, а при ее восстановлении — восстановить. Процесс сохранения и восстановления контекста задачи называется переключением контекста (context switching).

Немаловажным понятием является квант времени работы планировщика (tick) — это жестко фиксированный отрезок времени, в течение которого планировщик не вмешивается в выполнение задачи. По истечении кванта времени планировщик получает возможность приостановить текущую задачу и возобновить следующую, готовую к выполнению. Далее квант времени работы планировщика будет называться системным квантом. Для отсчета системных квантов в МК обычно используется прерывание от таймера/счетчика. Системный квант используется как единица измерения интервалов времени средствами ОСРВ.

Уменьшая продолжительность системного кванта, можно добиться более быстрой реакции программы на внешние события, однако это приведет к увеличению частоты вызова планировщика, что скажется на производительности вычислительной системы в целом.

Подводя итог, можно выделить три основные функции ядра любой ОСРВ:

1. Работа планировщика, благодаря которой создается эффект параллельного выполнения нескольких задач за счет быстрого переключения между ними.
2. Переключение контекста, благодаря которому выполнение одной задачи не сказывается на остальных задачах (задачи работают независимо).
3. Временная база, основанная на системном кванте как единице измерения времени.

Вышеприведенное описание основ ОСРВ является очень обобщенным. Существует еще целый ряд понятий, таких как приоритеты задач, средства синхронизации, передача информации между задачами и др., которые будет раскрыты позже на примере конкретной ОСРВ — FreeRTOS.

Соглашения о типах данных и именах идентификаторов

Как упоминалось в [1], бóльшая (подавляющая) часть FreeRTOS написана на языке Си. Имена идентификаторов в исходном коде ядра и демонстрационных проектах подчиняются определенным соглашениям, зная которые проще понимать тексты программ [5].

Имена переменных и функций представлены в префиксной форме (так называемая Венгерская нотация): имена начинаются с одной или нескольких строчных букв — префикса.

Для переменных префикс определяет тип переменной согласно таблице 1.

Например, ***ulMemCheck*** — переменная типа unsigned long, ***pxCreatedTask*** — переменная типа «указатель на структуру».

API-функции FreeRTOS имеют префиксы, обозначающие тип возвращаемого значения, как и для переменных. Системные функции, область видимости которых ограничена файлом исходного кода ядра (то есть имеющие спецификатор static), имеют префикс ***prv***.

Следом за префиксом функции следует имя модуля (файла с исходным кодом), в котором она определена. Например,

Таблица 1. Префиксы переменных

Префикс переменной	Ее тип
c	char
s	short
l	long
f	float
d	double
v	void
e	Перечисляемый тип (enum)
x	Структуры (struct) и др. типы
p	Указатель (дополнительно к вышеперечисленным)
u	Беззнаковый (дополнительно к вышеперечисленным)

vTaskStartScheduler() — функция, возвращающая тип void, которая определена в файле ***task.c***, ***uxQueueMessagesWaiting()*** — возвращает некий беззнаковый целочисленный тип, определена в файле ***queue.c***.

Встроенные типы данных (short, char и т.д.) не используются в исходном коде ядра. Вместо этого используется набор специальных типов, которые определены индивидуально для каждого порта в файле ***portmacro.h*** и начинаются с префикса port. Список специальных типов FreeRTOS приведен в таблице 2.

Таблица 2. Специальные типы FreeRTOS

Специальный тип FreeRTOS	Соответствующий встроенный тип
portCHAR	char
portSHORT	short
portLONG	long
portTickType	Тип счетчика системных квантов
portBASE_TYPE	Наиболее употребительный тип во FreeRTOS

Это сделано для обеспечения независимости кода ядра от конкретных компилятора и МК. В демонстрационных проектах так же использованы только специальные типы FreeRTOS, однако в своих проектах можно использовать встроенные типы данных. Это окажется полезным для разграничения идентификаторов, относящихся к ядру FreeRTOS, от идентификаторов, использующихся в прикладных задачах. Напротив, использование типов данных FreeRTOS позволит добиться большей кроссплатформенности создаваемого кода.

Подробнее следует остановиться на типах ***portTickType*** и ***portBASE_TYPE***:

1. ***portTickType*** может быть целым беззнаковым 16- или 32-битным. Он определяет тип системной переменной, которая используется для подсчета количества системных квантов, прошедших с момента старта планировщика. Таким образом, ***portTickType*** задает максимальный временной интервал, который может быть отсчитан средствами FreeRTOS. В случае 16-битного ***portTickType*** максимальный интервал составляет 65 536 квантов, в случае 32-битного — 4 294 967 296 квантов. Использование 16-битного счетчика квантов оправдано на 8- и 16-битных платформах, так как позволяет значительно повысить их быстродействие.
2. ***portBASE_TYPE*** определяет тип, активно используемый в коде ядра FreeRTOS. Операции с типом ***portBASE*** должны выполняться как можно более эффективно на данном МК, поэтому разрядность типа ***portBASE_TYPE*** устанавливается идентичной разрядности целевого МК. Например, для 8-битных МК это будет char, для 16-битных — short.

Идентификаторы макроопределений также начинаются с префикса, который определяет, в каком файле этот макрос находится (табл. 3).

Таблица 3. Префиксы макросов, используемых в FreeRTOS

Префикс	Где определен	Пример макроопределения
port	portable.h	portMAX_DELAY
tsk, task	task.h	taskENTER_CRITICAL()
pd	projdefs.h	pdTRUE
config	FreeRTOSConfig.h	configUSE_PREEMPTION
err	projdefs.h	errQUEUE_FULL

Задачи

Любая программа, которая выполняется под управлением FreeRTOS, представляет собой множество отдельных независимых задач. Каждая задача выполняется в своем собственном контексте без случайных зависимостей от других задач и ядра FreeRTOS. Только одна задача из множества может выполняться в один момент времени, и планировщик ответственен, какая именно. Планировщик останавливает и возобновляет выполнение всех задач по очереди, чтобы достичь эффекта одновременного выполнения нескольких задач на одном процессоре. Так как задача «не знает» об активности планировщика, то он отвечает за переключение контекста при смене выполняющейся задачи. Для достижения этого каждая задача имеет свой собственный стек. При смене задачи ее контекст сохраняется в ее собственном стеке, что позволяет восстановить контекст при возобновлении задачи [4].

Как было сказано выше, при грубом приближении задача может находиться в двух состояниях: выполняться и не выполняться. При подробном рассмотрении состояние «задача не выполняется» подразделяется на несколько различных состояний в зависимости от того, как она была остановлена (рис. 5).

Подробно рассмотрим состояния задачи в FreeRTOS. Говорят, что задача выполняется (running), если в данный момент времени процессор занят ее выполнением. Состояние готовности (ready) характеризует задачу, готовую к выполнению, но не выполняющуюся, так как в данный момент времени процессор занят выполнением другой задачи. Готовые к выполнению задачи (с одинаковым приоритетом) по очереди переходят в состояние выполнения и пребывают в нем в течение одного системного кванта, после чего возвращаются в состояние готовности.

Задача находится в блокированном состоянии, если она ожидает наступления временного или внешнего события (event). Например, вызвав API-функцию ***vTaskDelay()***, задача переведет себя в блокированное состояние до тех пор, пока не пройдет временной период задержки (delay): это будет временное событие. Задача блокирована, если она ожидает события, связанного с другими объектами ядра — очередями и семафорами: это будет внешнее (по отношению к задаче) событие. Нахождение задачи в блокированном состоянии ограниче-

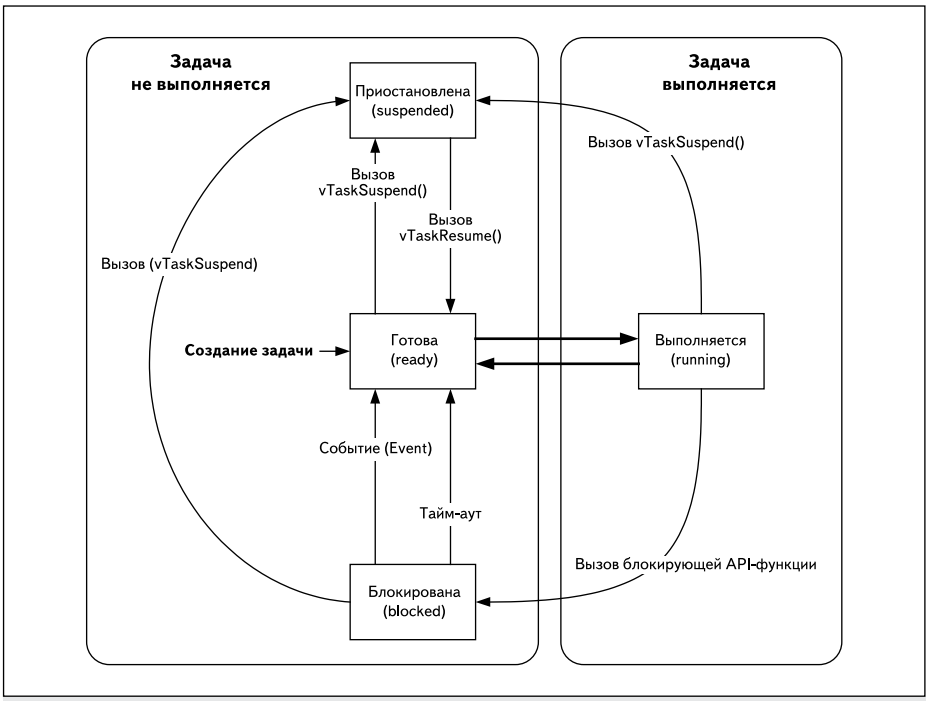

Рис. 5. Состояния задачи в FreeRTOS

но тайм-аутом. То есть если ожидаемое внешнее событие не наступило в течение тайм-аута, то задача возвращается в состояние готовности к выполнению. Это предотвращает «подвисание» задачи при ожидании внешнего события, которое по каким-то причинам никогда не наступит. Блокированная задача не получает процессорного времени.

Приостановленная (suspended) задача также не получает процессорного времени, однако, в отличие от блокированного состояния, переход в приостановленное состояние и выход из него осуществляется в явном виде вызовом API-функций ***vTaskSuspend()*** и ***xTaskResume()***. Тайм-аут для приостановленного состояния не предусмотрен, и задача может оставаться приостановленной сколь угодно долго [5].

В любой программе реального времени есть как менее, так и более ответственные задачи. Под «ответственностью» задачи здесь понимается время реакции программы на внешнее событие, которое обрабатывается задачей. Например, ко времени реакции на срабатывание датчика в производственной установке предъявляются куда более строгие требования, чем ко времени реакции на нажатие клавиши на клавиатуре. Для обеспечения преимущества на выполнение более ответственных задач во FreeRTOS применяется механизм приоритетов задач (Task priorities).

Среди всех задач, находящихся в состоянии готовности, планировщик отдаст управление той задаче, которая имеет наивысший приоритет. Задача будет выполняться до тех пор, пока она не будет блокирована или приостановлена или пока не появится готовая к выполнению задача с более высоким приоритетом.

Каждой задаче назначается приоритет от 0 до (***configMAX_PRIORITIES*** — 1). Меньшее значение приоритета соответствует меньшему приоритету. Наиболее низкий приоритет у задачи «бездействие», значение которого определено в ***tskIDLE_PRIORITY*** как 0. Изменяя значение ***configMAX_PRIORITIES***, можно определить любое число возможных приоритетов, однако уменьшение ***configMAX_PRIORITIES*** позволяет уменьшить объем ОЗУ, потребляемый ядром.

Задачи в FreeRTOS реализуются в виде Си-функций. Обязательное требование к функции, реализующей задачу: она должна иметь один аргумент типа указатель на void и ничего не возвращать (void). Указатель на такую функцию определен как ***pdTASK_CODE***. Каждая задача — это небольшая программа со своей точкой входа, которая содержит бесконечный цикл:

```
void ATaskFunction( void *pvParameters )
    {
    /* Переменные могут быть объявлены здесь, как и в обычной
    функции. Каждый экземпляр этой задачи будет иметь свою
    собственную копию переменной iVariableExample. Если
    объявить переменную со спецификатором static, то будет
    создана только одна переменная iVariableExample,
    доступная из всех экземпляров задачи */
    int iVariableExample = 0;
    /* Тело задачи реализовано как бесконечный цикл */
    for( ;; )
    {
        /* Код, реализующий функциональность задачи */
    }
    /* Если все-таки произойдет выход из бесконечного цикла,
    то задача должна быть уничтожена ДО конца функции.
    Параметр NULL обозначает, что уничтожается задача,
    вызывающая API-функцию vTaskDelete() */
    vTaskDelete( NULL );
}
```

Задачи создаются API-функцией ***xTaskCreate()***, а уничтожаются ***xTaskDelete()***. Функция ***xTaskCreate()*** является одной из наиболее сложных API-функций. Ее прототип:

```
portBASE_TYPE xTaskCreate(pdTASK_CODE pvTaskCode,
                          const signed portCHAR * const pcName,
                          unsigned portSHORT usStackDepth,
                          void *pvParameters,
                          unsigned portBASE_TYPE uxPriority,
                          xTaskHandle *pxCreatedTask
);
```

xTaskCreate() в случае успешного создания задачи возвращает pdTRUE. Если же объема памяти кучи недостаточно для размещения служебных структур данных и стека задачи, то ***xTaskCreate()*** возвращает ***errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY***. Функции ***xTaskCreate()*** передаются следующие аргументы:

1. ***pvTaskCode*** — указатель на функцию, реализующую задачу (фактически — идентификатор функции в программе).
2. ***pcName*** — нуль-терминальная (заканчивающаяся нулем) строка, определяющая имя функции. Ядром не используется, а служит лишь для наглядности при отладке.
3. ***usStackDepth*** — глубина (размер) собственного стека создаваемой задачи. Размер задается в словах, хранящихся в стеке, а не в байтах. Например, если стек хранит 32-битные слова, а значение ***usStackDepth*** задано равным 100, то для размещения стека задачи будет выделено 4×100 = 400 байт. Размер стека в байтах не должен превышать максимального значения для типа ***size_t***. Размер стека, необходимый для корректной работы задачи, которая ничего не делает (содержит только пустой бесконечный цикл, как задача ***ATaskFunction*** выше), задается макросом ***configMINIMAL_STACK_SIZE***. Не рекомендуется создавать задачи с меньшим размером стека. Если же задача потребляет большие объемы стека, то необходимо задать большее значение ***usStackDepth***. Нет простого способа определить размер стека, необходимого задаче. Хотя возможен точный расчет, большинство программистов находят золотую середину между требованиями выделения достаточного размера стека и эффективного расхода памяти. Существуют встроенные механизмы экспериментальной оценки объема используемого стека, например API-функция ***uxTaskGetStackHighWaterMark()***. О возможностях контроля переполнения стека будет рассказано позже.
4. ***pvParameters*** — произвольный параметр, передаваемый задаче при ее создании. Задается в виде указателя на void, в теле задачи может быть преобразован в указатель на любой другой тип. Передача параметра оказывается полезной возможностью при создании нескольких экземпляров одной задачи.
5. ***uxPriority*** — определяет приоритет создаваемой задачи. Нуль соответствует самому низкому приоритету, (***configMAX_PRIORITIES*** — 1) — наивысшему. Значение аргумента ***uxPriority*** большее, чем (***configMAX_PRIORITIES*** — 1), приведет к назначению задаче приоритета (***configMAX_PRIORITIES*** — 1).
6. ***pxCreatedTask*** — может использоваться для получения дескриптора (handle) создаваемой задачи, который помещается по адресу ***pxCreatedTask*** после успешного создания задачи. Дескриптор можно использовать в дальнейшем для различных операций над задачей, например изменения приоритета задачи или ее уничтожения. Если в получении дескриптора нет необходимости, то ***pxCreatedTask*** должен быть установлен в NULL.

По сложившейся традиции первая программа в учебнике по любому языку программирования для компьютеров выводит на экран монитора фразу “Hello, world!”. Рискнем предположить, что для микроконтроллеров первая программа должна переключать логический уровень на своих выводах с некоторой частотой (проще говоря, мигать светодиодами).

Что ж, пришло время написать первую программу под управлением FreeRTOS. Программа будет содержать две задачи. Задача 1 будет переключать логический уровень на одном выводе МК, задача 2 — на другом. Частота переключения для разных выводов будет разной.

В качестве аппаратной платформы будет использоваться МК AVR ATmega128L, установленный на мезонинный модуль WIZ200WEB фирмы WIZnet (рис. 6) [7]. Как отправная точка будет взят демонстрационный проект, компилятор — WinAVR, версия 2010.01.10.

Прежде всего необходимо загрузить и установить компилятор WinAVR [8]. Далее с официального сайта [9] загрузить дистрибутив FreeRTOS и распаковать в удобное место (в статье это C:/).

Демонстрационный проект располагается в ***C:/FreeRTOSV6.1.0/Demo/AVR_ATMega323_WinAVR/*** и предназначен для выполнения на МК ATmega323. Файл ***makefile***, находящийся в директории проекта, содержит все настройки и правила компиляции и, в том числе, определяет, для какого МК компилируется проект. Для того чтобы целевой платформой стал МК ATmega128, необходимо в файле ***makefile*** отыскать строку:

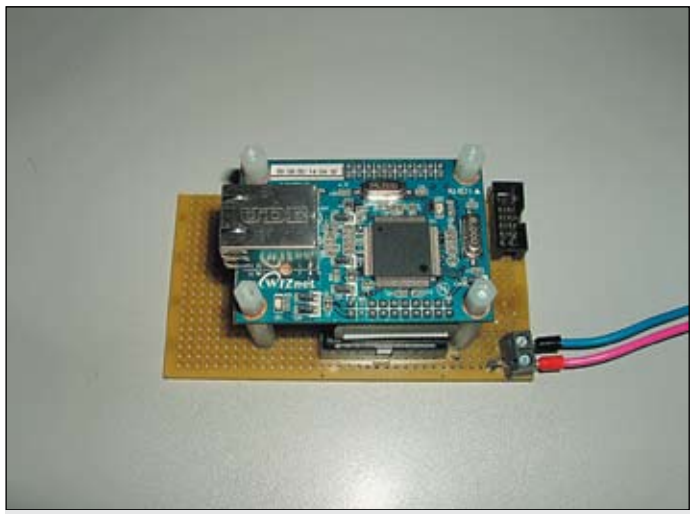

Рис. 6. Мезонинный модуль WIZ200WEB

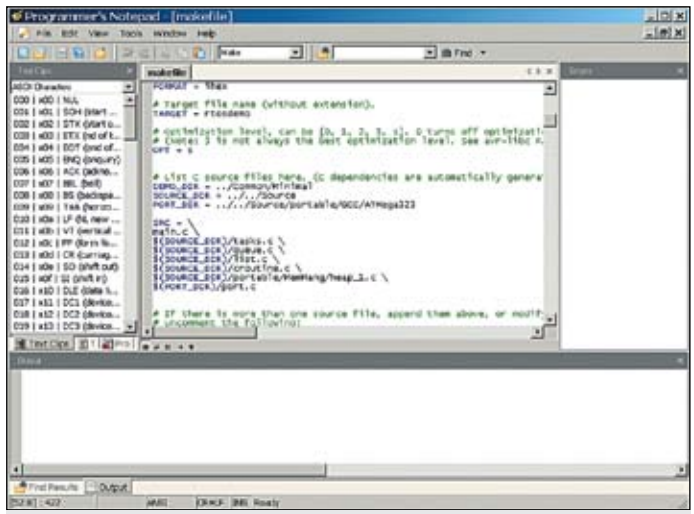

Рис. 7. Окно редактора Programmers Notepad

```
MCU = atmega323
```

и заменить ее на

```
MCU = atmega128
```

Для редактирования файлов можно применить простой, но удобный текстовый редактор Programmers Notepad, который поставляется вместе с компилятором WinAVR (рис. 7), запустить который можно выполнив ***Пуск → Все программы → WinAVR-20100110 → Programmers Notepad [WinAVR]*** или ***C:/WinAVR-20100110/pn/pn.exe*** (в случае установки WinAVR на диск C:/). Помимо прочего Programmers Notepad позволяет производить сборку (buid) проекта прямо из окна редактора.

Далее необходимо исключить из компиляции большинство исходных файлов проекта, отвечающих за демонстрацию всех возможностей FreeRTOS, оставив лишь основной файл ***main.c***. То есть заменить фрагмент файла ***makefile***:

```
SRC = \
main.c \
ParTest/ParTest.c \
serial/serial.c \
regtest.c \
$(SOURCE_DIR)/tasks.c \
$(SOURCE_DIR)/queue.c \
$(SOURCE_DIR)/list.c \
$(SOURCE_DIR)/croutine.c \
$(SOURCE_DIR)/portable/MemMang/heap_1.c \
$(PORT_DIR)/port.c \
$(DEMO_DIR)/crflash.c \
$(DEMO_DIR)/integer.c \
$(DEMO_DIR)/PollQ.c \
$(DEMO_DIR)/comtest.c
```

на:

```
SRC = \
main.c \
$(SOURCE_DIR)/tasks.c \
$(SOURCE_DIR)/queue.c \
$(SOURCE_DIR)/list.c \
$(SOURCE_DIR)/croutine.c \
$(SOURCE_DIR)/portable/MemMang/heap_1.c \
$(PORT_DIR)/port.c
```

Подготовительный этап закончен. Теперь можно переходить к редактированию файла ***main.c***. Его содержимое должно принять вид:

```
#include <stdlib.h>
#include <string.h>

#ifdef GCC_MEGA_AVR
   /* EEPROM routines used only with the WinAVR compiler. */
   #include <avr/eeprom.h>
#endif

/* Необходимые файлы ядра */
#include "FreeRTOS.h"
#include "task.h"
#include "croutine.h"

/*-----------------------------------------------------------*/
/* Функция задачи 1 */
void vTask1( void *pvParameters )
{
   /* Квалификатор volatile запрещает оптимизацию
   /* переменной ul */
   volatile unsigned long ul;
   /* Как и большинство задач, эта задача содержит
   /* бесконечный цикл */
   for( ;; )
   {
      /* Инвертировать бит 0 порта PORTF */
      PORTF ^= (1 << PF0);
      /* Задержка на некоторый период T1*/
      for( ul = 0; ul < 4000L; ul++ )
      {
         /* Это очень примитивная реализация задержки,
         /* в дальнейших примерах будут использоваться
         /* API-функции */
      }
   }
   /* Уничтожить задачу, если произошел выход
   /* из бесконечного цикла (в данной реализации выход
   /* заведомо не произойдет) */
   vTaskDelete( NULL );
}

/*-----------------------------------------------------------*/
/* Функция задачи 2, подобная задаче 1 */
void vTask2( void *pvParameters )
{
   volatile unsigned long ul;
   for( ;; )
   {
      /* Инвертировать бит 1 порта PORTF */
      PORTF ^= (1 << PF1);
      /* Задержка на некоторый период T2*/
      for( ul = 0; ul < 8000L; ul++ )
      {
      }

   }
   vTaskDelete( NULL );
}

/*-----------------------------------------------------------*/
```

```
short main( void )
{

   /* Биты 0, 1 порта PORTF будут работать как ВЫХОДЫ */
   DDRF |= (1 << DDF0) | (1 << DDF1);

   /* Создать задачу 1, заметьте,  что реальная программа должна
   /* проверять возвращаемое  значение, чтобы убедиться,
   /* что задача создана успешно */
   xTaskCreate(vTask1,   /* Указатель на функцию,
                         /* реализующую задачу */
         (signed char *) "Task1",    /* Текстовое имя задачи.
                                     /* Только для наглядности */
         configMINIMAL_STACK_SIZE,  /* Размер стека –
                                     /* минимально необходимый*/
         NULL,     /* Параметр, передаваемый задаче, –
                   /* не используется */
         1,        /* Приоритет = 1 */
         NULL );   /* Получение дескриптора задачи – не используется */

   /* Создать задачу 2 */
   xTaskCreate( vTask2, ( signed char * ) "Task2",
configMINIMAL_STACK_SIZE, NULL, 1, NULL );

   /* Запустить планировщик. Задачи начнут выполняться. */
   vTaskStartScheduler();

   return 0;
}
/*-----------------------------------------------------------*/
```

Для сборки проекта из среды Programmers Notepad необходимо выбрать пункт меню ***Tools → [WinAVR] Make all*** (рис. 8). Сообщение об отсутствии ошибок (Errors: none) означает успешную сборку и получение файла прошивки ***rtosdemo.hex***, который должен появиться в директории проекта.

Используя любой программатор, необходимо загрузить файл прошивки в целевой МК. Автор использовал для этой цели аналог отладчика JTAG ICE (рис. 9). Возможна загрузка и через интерфейс SPI.

Подключив осциллограф к выводам 1, 2 разъема J2 — они подключены к выводам PF0 и PF1 ATmega128 соответственно (обозначены красным на рис. 9), можно наблюдать совместную работу двух независимых задач (рис. 10).

Рассмотрим подробнее, что происходит. Пусть после старта планировщик первой запустит задачу 1 (рис. 11). Она выполняется на протяжении 1 системного кванта времени, который задан равным 1 мс в файле

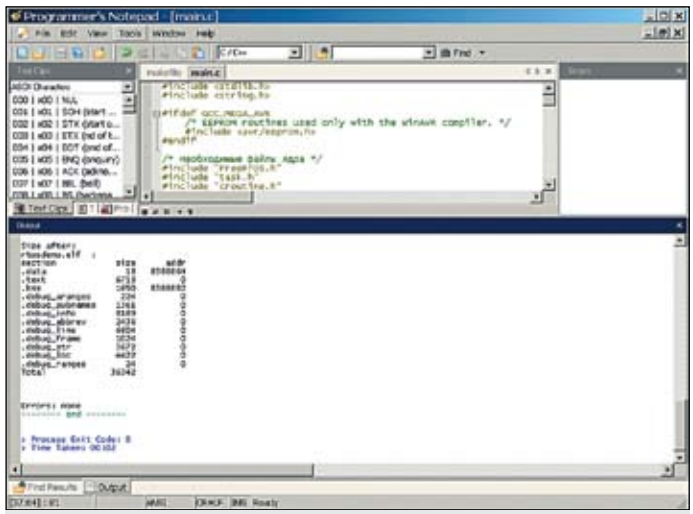

Рис. 8. Успешное завершение сборки проекта

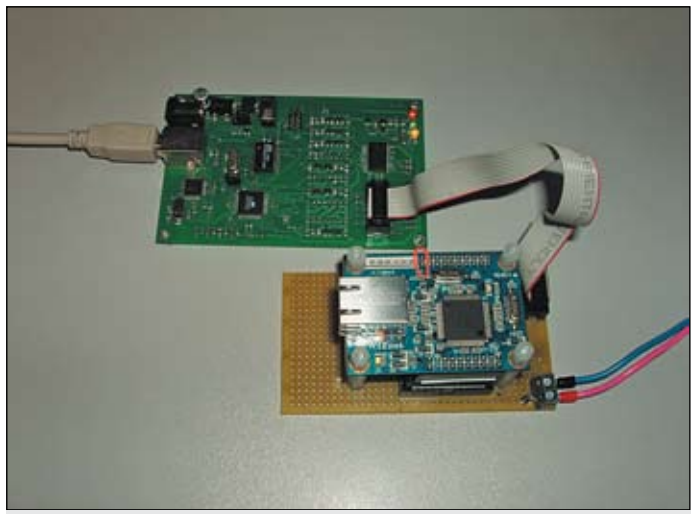

Рис. 9. Загрузка файла прошивки

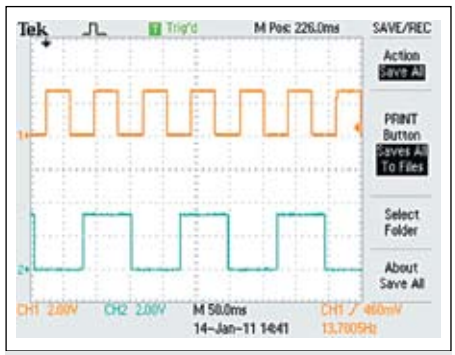

Рис. 10. Напряжение на выводах PF0 и PF1 ATmega128L (сверху вниз), полученное цифровым осциллографом

FreeRTOSConfig.h. В это время задача 2 находится в состоянии готовности. После чего вызывается планировщик, который переводит задачу 1 в состояние готовности, а задачу 2 — в состояние выполнения, так как задачи имеют одинаковый приоритет и задача 1 уже отработала один квант времени.

Пока выполняется задача 1, она увеличивает свой счетчик ***ul***. Когда планировщик переводит задачу 1 в состояние готовности, переменная ***ul*** сохраняется в собственном стеке задачи 1 и не увеличивается, пока выполняется задача 2. Как только переменная ***ul*** достигает значения 4000, она обнуляется (момент времени t1), а логический уровень на выводе PF0 инвертируется, однако это может произойти только в течение кванта времени выполнения задачи 1. Аналогично ведет себя задача 2, но ее счетчик обнуляется по достижении значения 8000. Таким образом, эта простейшая программа генерирует меандр с «плавающим» полупериодом, а разброс продолжительности полупериода достигает одного системного кванта, то есть 1 мс.

Выводы

В статье были рассмотрены основные принципы, заложенные во все ОСРВ. Описаны соглашения об именах идентификаторов и типах данных, используемых в исходном коде ядра FreeRTOS. Большое внимание уделено задаче как базовой единице программы для FreeRTOS. Подробно рассмотрены состояния задачи, дано объяснение понятию приоритета задачи. Описана API-функция создания задачи ***xTaskCreate()***. Приведен пример наипростейшей программы, выполняющейся под управлением FreeRTOS, приведены результаты тестирования и описаны происходящие процессы без углубления во внутреннюю реализацию FreeRTOS.

В следующих публикациях будет продолжено рассмотрение задач. Подробно будет рассказано о приоритетах задач, показано, каким образом можно менять приоритеты во время выполнения программы. Внимание будет уделено правильному способу приостанавливать задачи на заданное время и формировать задержки. Будет рассказано о задаче «бездействие» и о функции, вызываемой каждый системный квант времени. Будет показано, как правильно уничтожать задачи. Весь материал будет снабжен подробными примерами.

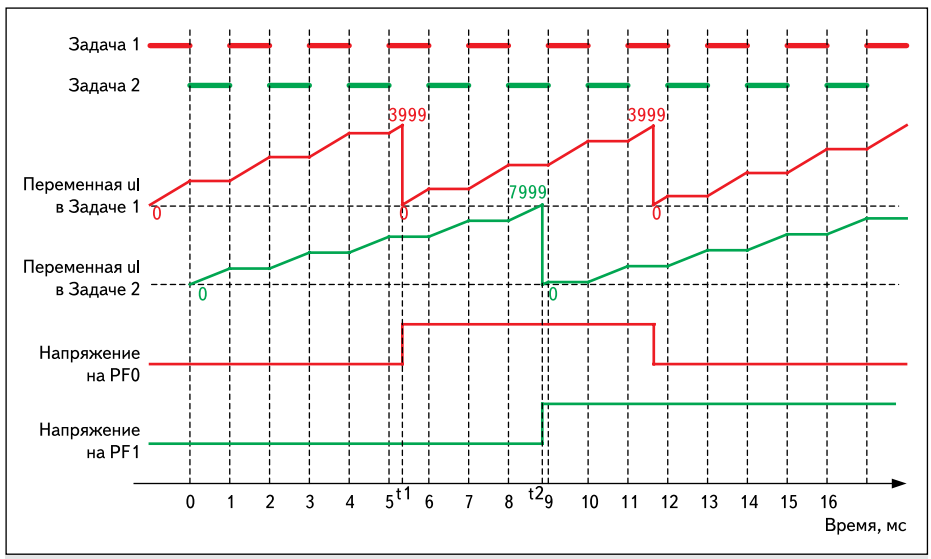

Рис. 11. Работа программы во времени

Литература

1. Курниц А. FreeRTOS — операционная система для микроконтроллеров // Компоненты и технологии. 2011. № 2.
2. http://www.freertos.org/implementation/index.html
3. http://www.freertos.org/a00015.html
4. http://www.freertos.org/taskandcr.html
5. http://www.freertos.org/a00017.html
6. Barry R. Using the freertos real time kernel: A Practical Guide. 2009.
7. http://www.wiznet.co.kr/Sub_Modules/en/product/Product_Detail.asp? cate1=5&cate2=44&cate3=0&pid=1025
8. http://winavr.sourceforge.net/download.html
9. http://sourceforge.net/projects/freertos/files/FreeRTOS/