FreeRTOS — операционная система для микроконтроллеров

Это первая статья из цикла, посвященного операционной системе для микроконтроллеров FreeRTOS. Статья познакомит читателя с задачами, которые решают операционные системы (ОС) для микроконтроллеров (МК). Освещены вопросы целесообразности применения, преимущества и недостатки, присущие ОС для МК. Представлены возможности FreeRTOS, описаны ее особенности, а также приведена структура дистрибутива FreeRTOS с кратким описанием назначения входящих в него файлов и директорий.

Андрей КУРНИЦ
kurnits@stim.by

Что такое ОС для МК?

В нынешний век высоких технологий все профессионалы знакомы с термином «операционная система» (ОС). История ОС начинается с 1960-х годов. Первые ОС предназначались для больших ЭВМ, а впоследствии — и для персональных компьютеров. Назначением ОС стало заполнение ниши между низкоуровневой аппаратурой и высокоуровневыми программами, они предоставляют программам удобный интерфейс обращения к системным ресурсам, будь то процессорное время, память или устройства ввода/вывода. С тех пор технологии шагнули далеко вперед: целую вычислительную систему (процессор, память, устройства ввода/вывода) разместили на одном кристалле — появились микроконтроллеры (МК). В соответствии с древним изречением «Природа не любит пустоты» удачная концепция ОС не могла не быть применена и к микроконтроллерам. В настоящее время создано и развивается множество ОС, ориентированных на выполнение на МК [1, 6]. Однако МК как платформа для выполнения ОС имеет существенные отличия от современных компьютеров.

Прежде всего, МК работает в режиме реального времени, то есть время реакции микроконтроллерного устройства на внешнее событие должно быть строго меньше заданной величины и должно быть сопоставимо со скоростью протекания внешних процессов. Типичный пример: время реакции на срабатывание датчика давления в промышленной установке должно быть не более 5 мс, иначе произойдет авария. Таким образом, ОС для МК — это операционная система реального времени (ОСРВ). К ОСРВ предъявляются жесткие временные требования в отличие от распространенных ОС общего назначения (Windows, UNIX-подобные и др.).

Во-вторых, микроконтроллер, по сути, это однокристальный компьютер с сильно ограниченными аппаратными ресурсами, хотя диапазон выпускаемых МК по производительности и объемам памяти очень широк. Встречаются как «карлики», например 8-разрядный ATtiny10 с 6 выводами, 32 байт ОЗУ, 1 кбайт ПЗУ и производительностью 12×10^6 операций в секунду (12 MIPS), так и «гиганты», например 32-разрядный TMS320C28346 с 256 выводами, 512 кбайт ОЗУ и производительностью 600×10^6 операций с плавающей точкой в секунду (600 MFLOPS). Тем не менее все МК имеют существенные аппаратные ограничения, что предъявляет специфические требования к ОСРВ для МК.

Их основные особенности:

1. Низкая производительность.
2. Малый объем ОЗУ и ПЗУ.
3. Отсутствие блока управления памятью (Memory management unit, MMU), используемого большинством современных ОС, например Windows и UNIX-подобными.
4. Отсутствие аппаратных средств поддержки многозадачности (например, средств быстрого переключения контекста).

В-третьих, микроконтроллер сам по себе предназначен для выполнения низкоуровневых задач, будь то опрос состояния кнопок, передача команды по I^2C-интерфейсу или включение обмотки электромотора. Программа для МК, как правило, обращается к периферии напрямую, программист имеет полный контроль над аппаратной частью, нет необходимости в посредниках между аппаратурой и прикладной программой. Может показаться, что операционная система для МК вообще не нужна, что любую программу можно написать и без ОС. На самом деле так оно и есть! Но есть один нюанс: микроконтроллер редко используют только для опроса состояния кнопок, только для передачи команды по I^2C-интерфейсу или только для включения обмотки электромотора. Гораздо чаще из МК пытаются «выжать» все, на что он способен, а в микроконтроллерное устройство заложить все возможные функции. Количество функций-задач, одновременно выполняемых МК, может доходить до нескольких десятков. И вот тут-то и начинаются проблемы.

Как организовать мультизадачность и поочередное выполнение каждой задачи? Как обеспечить запуск задачи через строго определенные интервалы времени? Как передать информацию от одной задачи другой? Обычно эти вопросы не встают перед программистом в начале разработки, а возникают где-то в середине, когда он запрограммировал большинство функций будущего устройства, используя изобретенные им самим средства «многозадачности». И тогда заказчик «просит» добавить еще несколько «маленьких» деталей в работу устройства, например сбор статистики работы и запись ее на какой-нибудь носитель… Знакомая ситуация?

Преимущества ОСРВ для МК

Здесь на помощь приходит ОСРВ. Рассмотрим преимущества, которые получил бы наш гипотетический программист, заложив в основу программного обеспечения своего устройства ОСРВ:

1. Многозадачность. ОСРВ предоставляет программисту готовый, отлаженный механизм многозадачности. Теперь каждую отдельную задачу можно программировать по отдельности так, как будто остальных задач не существует. Например, можно разработать архитектуру программы, то есть разбить ее на отдельные задачи и распределить их между командой программистов. Программисту не нужно заботиться о переключении между задачами: за него это сделает ОСРВ в соответствии с алгоритмом работы планировщика.

2. Временная база. Необходимо отмерять интервалы времени? Пожалуйста, любая ОСРВ имеет удобный программный интерфейс для отсчета интервалов времени и выполнения каких-либо действий в определенные моменты времени.
3. Обмен данными между задачами. Необходимо передать информацию от одной задачи к другой без потерь? Используйте очередь, которая гарантирует, что сообщения дойдут до адресата в том объеме и в той последовательности, в которой были отправлены.
4. Синхронизация. Разные задачи обращаются к одному и тому же аппаратному ресурсу? Используйте мьютексы или критические секции для организации совместного доступа к ресурсам. Необходимо выполнять задачи в строгой последовательности или по наступлении определенного события? Используйте семафоры или сигналы для синхронизации задач.

Кроме этого, одна и та же ОСРВ для МК может выполняться на множестве архитектур микроконтроллеров. Какое преимущество это дает? Часто приходится решать задачу не как разработать устройство с требуемой функциональностью, а как перенести имеющуюся разработку на новую аппаратную платформу. Это может быть связано с завершением производства того или иного МК (окончание Life cycle), с появлением на рынке МК, включающего в состав некоторые блоки, которые ранее были реализованы как отдельные микросхемы, и т. д. В случае использования ОСРВ затраты времени и сил на переход на другую платформу будут заметно ниже за счет того, что часть кода, связанная с работой ОСРВ, останется без изменений. Изменения коснутся только кода, отвечающего за обращение к встроенной периферии (таймеры, АЦП, последовательный приемопередатчик и т. д.).

Однако за все надо платить. Использование ОСРВ приводит к определенным накладным расходам. Это:

1. Дополнительный расход памяти программ для хранения ядра ОСРВ.
2. Дополнительный расход памяти данных для хранения стека каждой задачи, семафоров, очередей, мьютексов и других объектов ядра операционной системы.
3. Дополнительные затраты времени процессора на переключение между задачами.

Когда можно обойтись без ОСРВ для МК?

Конечно же, если вам необходимо разработать простейшее устройство, например индикатор температуры, который будет выполнять две функции: опрос датчика и индикацию на 7-сегментный светодиодный индикатор, то применение ОСРВ в таком устройстве будет непозволительным расточительством и приведет, в конечном счете, к удорожанию устройства.

В этом случае можно применить один из «традиционных» для МК способов организации многозадачности. Прежде всего, это циклический алгоритм (round robin) [3], когда программист помещает все задачи в тело бесконечного цикла. При этом на подпрограммы, реализующие задачи, накладываются следующие ограничения:

1. Подпрограмма не должна содержать циклов ожидания наступления какого-либо события, например прерывания.
2. Подпрограмма должна лишь проверять, наступило ли событие, и как можно быстрее передавать управление следующей подпрограмме, то есть завершать свое выполнение.
3. Подпрограмма должна сохранять свое текущее состояние (например, в статической или глобальной переменной) до следующего вызова.

Таким образом, каждая задача представляется в виде конечного автомата. Дальнейшее развитие эта идея получила в SWITCH-технологии программирования [4, 5].

Резюме

Итак, применение ОСРВ оправдано в случае использования достаточно мощного МК при разработке сложного устройства с множеством функций, например:

1. Опрос датчиков.
2. Интерфейс с пользователем (простейшие клавиатура и дисплей).
3. Выдача управляющего воздействия.
4. Обмен информацией по нескольким внутрисхемным шинам I^2C, SPI, 1Wire и др.
5. Обмен информацией с внешними устройствами по интерфейсам RS-232C, RS-485, CAN, Ethernet, USB и др.
6. Реализация высокоуровневых протоколов, например TCP/IP, ProfiBus, ModBus, CANOpen и др.
7. Поддержка Flash-накопителей и, соответственно, файловой системы.

Обзор FreeRTOS

FreeRTOS — это многозадачная, мультиплатформенная, бесплатная операционная система жесткого реального времени с открытым исходным кодом. FreeRTOS была разработана компанией Real Time Engineers Ltd. специально для встраиваемых систем. На момент написания статьи (версия FreeRTOS 6.1.0) ОС официально поддерживает 23 архитектуры и 57 платформ (в подавляющем большинстве — микроконтроллеры) [7]. В течение 2008 и 2009 годов произошло более 77 500 загрузок FreeRTOS с официального сайта, что делает ее одной из самых популярных ОСРВ на сегодня. Бóльшая часть кода FreeRTOS написана на языке Си, ассемблерные вставки минимального объема применяются лишь там, где невозможно применить Си из-за специфики конкретной аппаратной платформы.

Существуют так называемые официально поддерживаемые аппаратные платформы — официальные порты и неофициальные, которые поставляются «как есть» и не поддерживаются напрямую. Кроме того, для одного и того же порта могут поддерживаться несколько средств разработки. Список официальных портов и средств разработки приведен в таблице 1.

Основные характеристики FreeRTOS:

1. Планировщик FreeRTOS поддерживает три типа многозадачности:
   – вытесняющую;
   – кооперативную;
   – гибридную.
2. Размер ядра FreeRTOS составляет всего 4–9 кбайт, в зависимости от типа платформы и настроек ядра.
3. FreeRTOS написана на языке Си (исходный код ядра представлен в виде всего лишь четырех Си-файлов).
4. Поддерживает задачи (tasks) и сопрограммы (co-routines). Сопрограммы специально созданы для МК с малым объемом ОЗУ.
5. Богатые возможности трассировки.
6. Возможность отслеживать факт переполнения стека.
7. Нет программных ограничений на количество одновременно выполняемых задач.
8. Нет программных ограничений на количество приоритетов задач.
9. Нет ограничений в использовании приоритетов: нескольким задачам может быть назначен одинаковый приоритет.
10. Развитые средства синхронизации «задача – задача» и «задача – прерывание»:
    – очереди;
    – двоичные семафоры;
    – счетные семафоры;
    – рекурсивные семафоры;
    – мьютексы.
11. Мьютексы с наследованием приоритета.
12. Поддержка модуля защиты памяти (Memory protection unit, MPU) в процессорах Cortex-M3.
13. Поставляется с отлаженными примерами проектов для каждого порта и для каждой среды разработки.
14. FreeRTOS полностью бесплатна, модифицированная лицензия GPL позволяет использовать FreeRTOS в проектах без раскрытия исходных кодов.
15. Документация в виде отдельного документа платная, но на официальном сайте [7] в режиме on-line доступно исчерпывающее техническое описание на английском языке.

Работа планировщика FreeRTOS в режиме вытесняющей многозадачности имеет много общего с алгоритмом переключения потоков в современных ОС общего назначения. Вытесняющая многозадачность предполагает, что любая выполняющаяся задача с низким приоритетом прерывается готовой к выполнению задачей с более высоким приоритетом. Как только высокоприоритетная

Таблица 1. Список официальных портов FreeRTOS и средств разработки

Производитель	Поддерживаемые семейства (ядра)	Поддерживаемые средства разработки
Altera	Nios II	Nios II IDE, GCC
Atmel	SAM3 (Cortex-M3)	IAR, GCC, Keil, Rowley CrossWorks
	SAM7 (ARM7)	
	SAM9 (ARM9)	
	AT91	
	AVR32 UC3	
Cortus	APS3	Cortus IDE, GCC
Energy Micro	EFM32 (Cortex-M3)	IAR
Freescale	Coldfire V2	Codewarrior, GCC, Eclipse
	Coldfire V1	
	другие Coldfire	
	HCS12	
	PPC405 & PPC440	
Fujitsu	32 бит (например, MB91460)	Softune
	16 бит (например, MB96340 16FX)	
Luminary Micro / Texas Instruments	Все МК Stellaris на основе ядра Cortex-M3	Keil, IAR, Code Red, CodeSourcery GCC, Rowley CrossWorks
Microchip	PIC32	MPLAB C32, MPLAB C30, MPLAB C18, wizC
	PIC24	
	dsPIC	
	PIC18	
NEC	V850 (32 бит)	IAR
	78K0R (16 бит)	
NXP	LPC1700 (Cortex-M3)	GCC, Rowley CrossWorks, IAR, Keil, Red Suite, Eclipse
	LPC2000 (ARM7)	
Renesas	RX600/RX62N	GCC, HEW (High Performance Embedded Workbench), IAR
	SuperH	
	H8/S	
Silicon Labs (бывший Cygnal)	Сверхбыстрые i8051 совместимые МК	SDCC
ST	STM32 (Cortex-M3)	IAR, GCC, Keil, Rowley CrossWorks
	STR7 (ARM7)	
	STR9 (ARM9)	
Texas Instruments	MSP430	Rowley CrossWorks, IAR, GCC
Xilinx	PPC405, выполняющийся на Virtex4 FPGA	GCC
	PPC440, выполняющийся на Virtex5 FPGA	
	Microblaze	
i8086	Любой x86 совместимый процессор в реальном режиме (Real mode)	Open Watcom, Borland, Paradigm
	Win32 симулятор	Visual Studio

задача выполнила свои действия, она завершает свою работу или переходит в состояние ожидания, и управление снова получает задача с низким приоритетом. Переключение между задачами осуществляется через равные кванты времени работы планировщика, то есть высокоприоритетная задача, как только она стала готова к выполнению, ожидает окончания текущего кванта, после чего управление получает планировщик, который передает управление высокоприоритетной задаче.

Таким образом, время реакции FreeRTOS на внешние события в режиме вытесняющей многозадачности — не больше одного кванта времени планировщика, который можно задавать в настройках. По умолчанию он равен 1 мс.

Если готовы к выполнению несколько задач с одинаковым приоритетом, то в таком случае планировщик выделяет каждой из них по одному кванту времени, по истечении которого управление получает следующая задача с таким же приоритетом, и так далее по кругу.

Кооперативная многозадачность отличается от вытесняющей тем, что планировщик самостоятельно не может прервать выполнение текущей задачи, даже если появилась готовая к выполнению задача с более высоким приоритетом. Каждая задача должна самостоятельно передать управление планировщику. Таким образом, высокоприоритетная задача будет ожидать, пока низкоприоритетная завершит свою работу и отдаст управление планировщику. Время реакции системы на внешнее событие становится неопределенным и зависит от того, как долго текущая задача будет выполняться до передачи управления. Кооперативная многозадачность применялась в семействе ОС Windows 3.x.

Вытесняющая и кооперативная концепции многозадачности объединяются вместе в гибридной многозадачности, когда вызов планировщика происходит каждый квант времени, но, в отличие от вытесняющей многозадачности, программист имеет возможность сделать это принудительно в теле задачи. Особенно полезен этот режим, когда необходимо сократить время реакции системы на прерывание. Допустим, в текущий момент выполняется низкоприоритетная задача, а высокоприоритетная ожидает наступления некоторого прерывания. Далее происходит прерывание, но по окончании работы обработчика прерываний выполнение возвращается к текущей низкоприоритетной задаче, а высокоприоритетная ожидает, пока закончится текущий квант времени. Однако если после выполнения обработчика прерывания передать управление планировщику, то он передаст управление высокоприоритетной задаче, что позволяет значительно сократить время реакции системы на прерывание, связанное с внешним событием.

Для оценки затрат времени, вносимых планировщиком FreeRTOS, можно сравнить два распространенных семейства МК: PIC и AVR. Затраты времени складываются из времени переключения контекста, когда планировщик определяет задачу для выполнения в следующем кванте времени, и времени сохранения/восстановления контекста, когда текущее состояние задачи (регистры процессора) сохраняется/извлекается из стека (таблица 2). Замеры приведены для компиляторов MPLAB PIC18 compiler и WinAVR соответственно, уровень оптимизации — максимальный по скорости.

Для того чтобы оценить объем ОЗУ, требуемый для работы FreeRTOS, достаточно привести расчет расхода ОЗУ для следующей конфигурации:

1. Порт для процессоров ARM7, среда разработки IAR STR71x.
2. Полная оптимизация (Full optimization) включена.
3. Все компоненты FreeRTOS, кроме сопрограмм и трассировки, включены.
4. 4 приоритета задач.

Объемы расхода ОЗУ для такой конфигурации приведены в таблице 3.

Расход ОЗУ будет существенно ниже при работе FreeRTOS на 8- и 16-битных архитектурах.

Таблица 2. Расход времени на переключение между задачами

Микроконтроллер	Частота тактирования, МГц	Время переключения контекста, мкс	Время сохранения / восстановления контекста, мкс
ATMega323	8	41,8	~8
PIC18F452	20	66,2	~10

Таблица 3. Объемы ОЗУ, требуемые для работы FreeRTOS

Объект	Расход ОЗУ, байт
Планировщик (sheduler)	236
Каждая дополнительная очередь (queue)	76 + память для хранения всех элементов очереди (зависит от размера очереди)
Каждая дополнительная задача (task)	64 + стек задачи

Кроме самой FreeRTOS, существуют также ее коммерческие версии: SafeRTOS и OpenRTOS. SafeRTOS — это ОСРВ, соответствующая уровню функциональной безопасности SIL3, имеющая такую же функциональную модель, что и FreeRTOS, и ориентированная на применение в системах с высокими требованиями к безопасности, например в медицинской и аэрокосмической отраслях. OpenRTOS отличается от FreeRTOS

лишь тем, что поставляется под коммерческой лицензией, с гарантией производителя и отменяет некоторые несущественные ограничения, присущие FreeRTOS. Подробно с особенностями SafeRTOS и OpenRTOS можно ознакомиться в [8].

Конечно, FreeRTOS — это далеко не единственный выбор для разработчика. В настоящее время существует множество других ОСРВ для МК, среди которых можно назвать uC/OS-II, µClinux, Salvo, jacOS и др. [6]. Однако обсуждение достоинств и недостатков этих ОС выходит за рамки данной статьи.

С чего начать?

Начать разработку микроконтроллерного устройства, работающего под управлением FreeRTOS, можно с загрузки ее последней версии по адресу [9]. Дистрибутив FreeRTOS доступен в виде обычного или самораспаковывающегося ZIP-архива. Дистрибутив содержит непосредственно код ядра (в виде нескольких заголовочных файлов и файлов с исходным кодом) и демонстрационные проекты (по одному проекту на каждую среду разработки для каждого порта). Далее следует распаковать архив в любое подходящее место на станции разработки.

Несмотря на достаточно большое количество файлов в архиве (5062 файла для версии 6.1.0), структура директорий на самом деле проста. Если планируется проектировать устройства на 2–3 архитектурах в 1–2 средах разработки, то бóльшая часть файлов, относящихся к демонстрационным проектам и различным средам разработки, не понадобится.

Подробная структура директорий приведена на рисунке.

Весь исходный код ядра находится в директории ***/Source***. Его составляют следующие файлы:

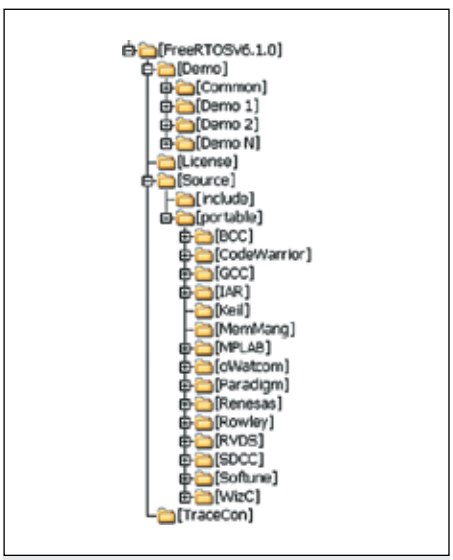

Рисунок. Структура директорий FreeRTOS после установки на станцию разработки

1. ***tasks.c*** — планировщик, реализация механизма задач.
2. ***queue.c*** — реализация очередей.
3. ***list.c*** — внутренние нужды планировщика, однако функции могут использоваться и в прикладных программах.
4. ***croutine.c*** — реализация сопрограмм (может отсутствовать в случае, если сопрограммы не используются).

Заголовочные файлы, которые находятся в директории ***Source/Include***:

1. ***tasks.h***, ***queue.h***, ***list.h***, ***croutine.h*** — заголовочные файлы соответственно для одноименных файлов с кодом.
2. ***FreeRTOS.h*** — содержит препроцессорные директивы для настройки компиляции.
3. ***mpu_wrappers.h*** — содержит переопределения функций программного интерфейса (API-функций) FreeRTOS для поддержки модуля защиты памяти (MPU).
4. ***portable.h*** — платформенно-зависимые настройки.
5. ***projdefs.h*** — некоторые системные определения.
6. ***semphr.h*** — определяет API-функции для работы с семафорами, которые реализованы на основе очередей.
7. ***StackMacros.h*** — содержит макросы для контроля переполнения стека.

Каждая аппаратная платформа требует небольшой части кода ядра, которая реализует взаимодействие FreeRTOS с этой платформой. Весь платформенно-зависимый код находится в поддиректории ***/Source/Portable***, где он систематизирован по средам разработки (IAR, GCC и т. д.) и аппаратным платформам (например, AtmelSAM7S64, MSP430F449). К примеру, поддиректория ***/Source/Portable/GCC/ATMega323*** содержит файлы port.c и portmacro.h, реализующие сохранение/восстановление контекста задачи, инициализацию таймера для создания временной базы, инициализацию стека каждой задачи и другие аппаратно-зависимые функции для микроконтроллеров семейства mega AVR и компилятора WinAVR (GCC).

Отдельно следует выделить поддиректорию ***/Source/Portable/MemMang***, в которой содержатся файлы ***heap_1.c***, ***heap_2.c***, ***heap_3.c***, реализующие 3 различных механизма выделения памяти для нужд FreeRTOS, которые будут подробно описаны позже.

В директории ***/Demo*** находятся готовые к компиляции и сборке демонстрационные проекты (***Demo 1***, ***Demo 2***, …, ***Demo N*** на рисунке). Общая часть кода для всех демонстрационных проектов выделена в поддиректорию ***/Demo/Common***.

Чтобы использовать FreeRTOS в своем проекте, необходимо включить в него файлы исходного кода ядра и сопутствующие заголовочные файлы. Нет необходимости модифицировать их или понимать их реализацию.

Например, если планируется использовать порт для микроконтроллеров MSP430 и GCC-компилятор, то для создания проекта «с нуля» понадобятся поддиректории ***/Source/Portable/GCC/MSP430F449*** и ***/Source/Portable/MemMang***. Все остальные поддиректории из директории ***/Source/Portable*** не нужны и могут быть удалены.

Если же планируется модифицировать существующий демонстрационный проект (что, собственно, и рекомендуется сделать в начале изучения FreeRTOS), то понадобятся также поддиректории ***/Demo/msp430_GCC*** и ***/Demo/Common***. Остальные поддиректории, находящиеся в ***/Demo***, не нужны и могут быть удалены.

При создании приложения рекомендуется использовать ***makefile*** (или файл проекта среды разработки) от соответствующего демонстрационного проекта как отправную точку. Целесообразно исключить из сборки (***build***) файлы из директории ***/Demo***, заменив их своими, а файлы из директории ***/Source*** оставить нетронутыми. Это гарантия того, что все исходные файлы ядра FreeRTOS будут включены в сборку и настройки компилятора останутся корректными.

Следует упомянуть также о заголовочном файле ***FreeRTOSConfig.h***, который находится в каждом демонстрационном проекте. ***FreeRTOSConfig.h*** содержит определения (***#define***), позволяющие произвести настройку ядра FreeRTOS:

1. Набор системных функций.
2. Использование сопрограмм.
3. Количество приоритетов задач и сопрограмм.
4. Размеры памяти (стека и кучи).
5. Тактовая частота МК.
6. Период работы планировщика — квант времени, выделяемый каждой задаче для выполнения, который обычно равен 1 мс.

Отключение некоторых системных функций и уменьшение количества приоритетов позволяет уменьшить расход памяти программ и данных.

В дистрибутив FreeRTOS включены также средства для конвертирования трассировочной информации, полученной от планировщика, в текстовую форму (директория ***/TraceCon***) и текст лицензии (директория ***/License***).

Выводы

С помощью первой статьи цикла читатель мог познакомиться с операционной системой для микроконтроллеров FreeRTOS. Показаны ее основные особенности. Описано содержимое дистрибутива FreeRTOS. Приведены основные шаги, с которых следует начинать разработку устройства, работающего под управлением FreeRTOS.

В следующих публикациях внимание будет уделено механизму многозадачности, а именно задачам и сопрограммам. Будет приведен образец работы планировщика на примере микроконтроллеров AVR фирмы Atmel и компилятора WinAVR (GCC).

Литература

1. Сорокин С. Как много ОСРВ хороших… // Современные технологии автоматизации. 1997. № 2.
2. Борисов-Смирнов А. Операционные системы реального времени для микроконтроллеров // Chip news. 2008. № 5.
3. Сорокин С. Системы реального времени // Современные технологии автоматизации. 1997. № 2.
4. http://ru.wikipedia.org/wiki/Switch-технология
5. Татарчевский В. Применение SWITCH- технологии при разработке прикладного программного обеспечения для микроконтроллеров // Компоненты и технологии. 2006. № 11.
6. http://ru.wikipedia.org/wiki/Список_операционных_систем
7. http://www.freertos.org
8. http://www.freertos.org/index.html? http://www.freertos.org/a00114.html
9. http://sourceforge.net/projects/freertos/files/FreeRTOS/

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.

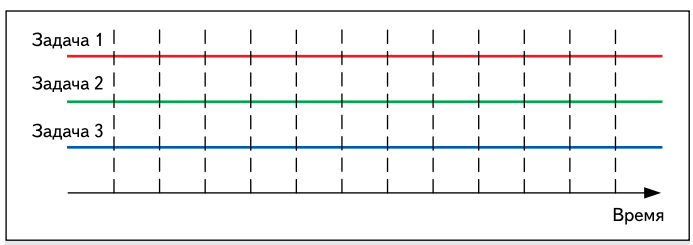

Рис. 1. Истинно параллельное выполнение задач

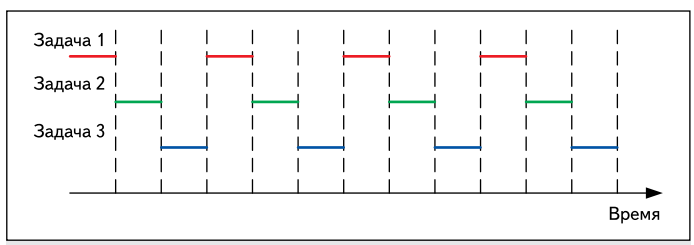

Рис. 2. Распределение процессорного времени между несколькими задачами в ОСРВ

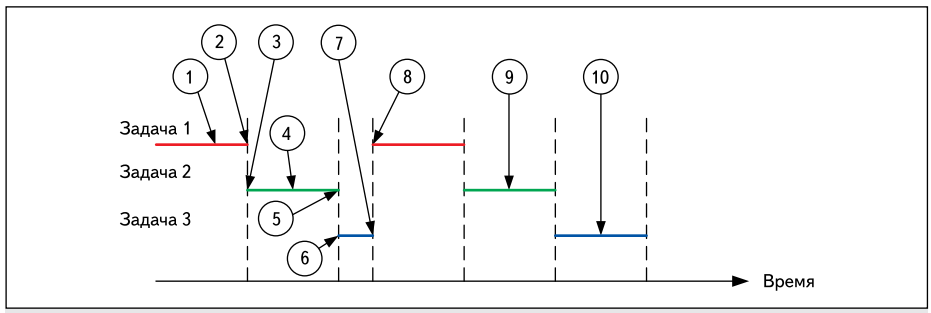

Рис. 3. Переключение между задачами, которые используют один и тот же аппаратный ресурс

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).

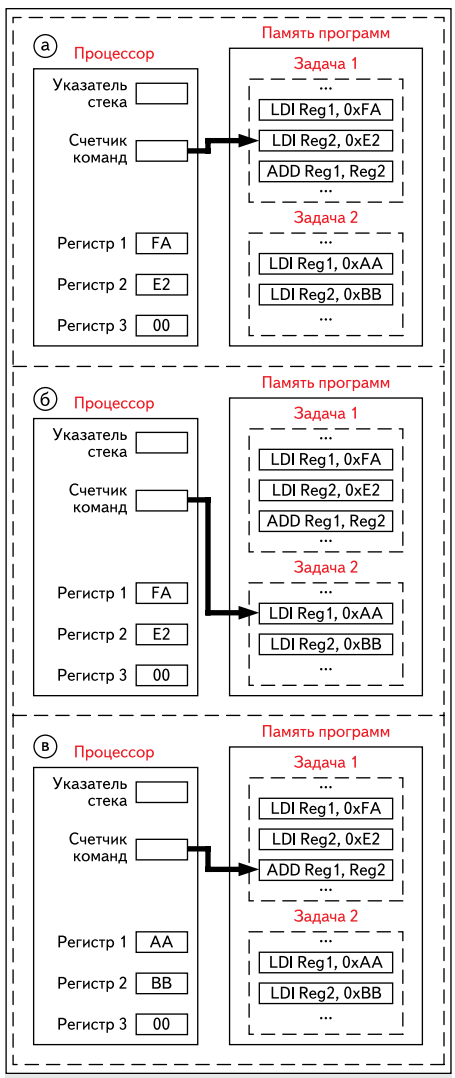

Рис. 4. Переключение между задачами без переключения контекста

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-

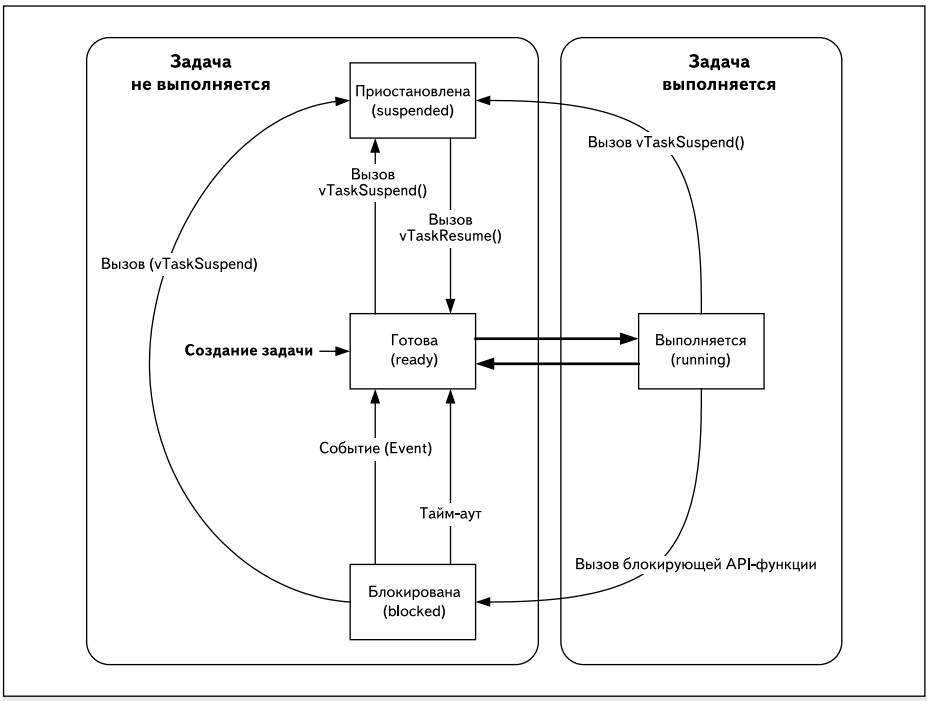

Рис. 5. Состояния задачи в FreeRTOS

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:

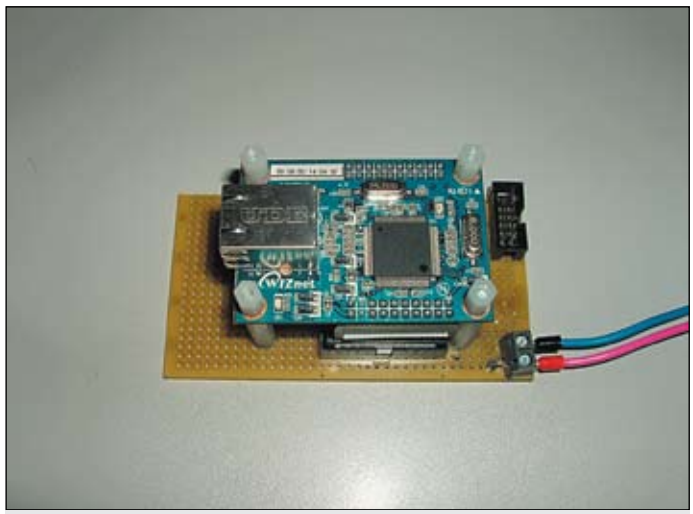

Рис. 6. Мезонинный модуль WIZ200WEB

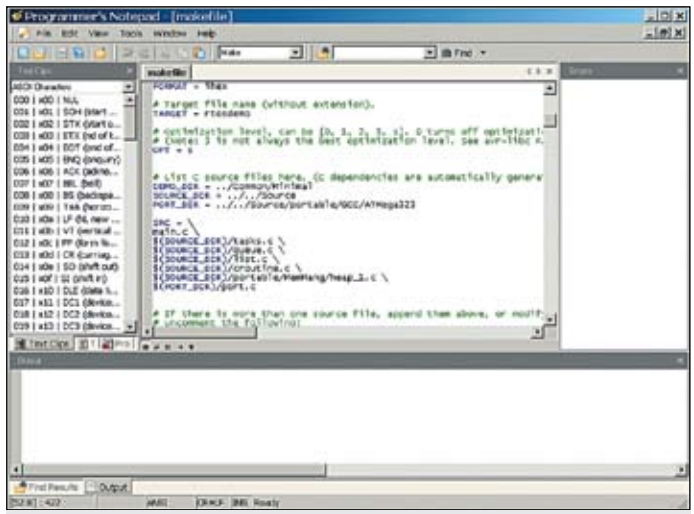

Рис. 7. Окно редактора Programmers Notepad

```
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

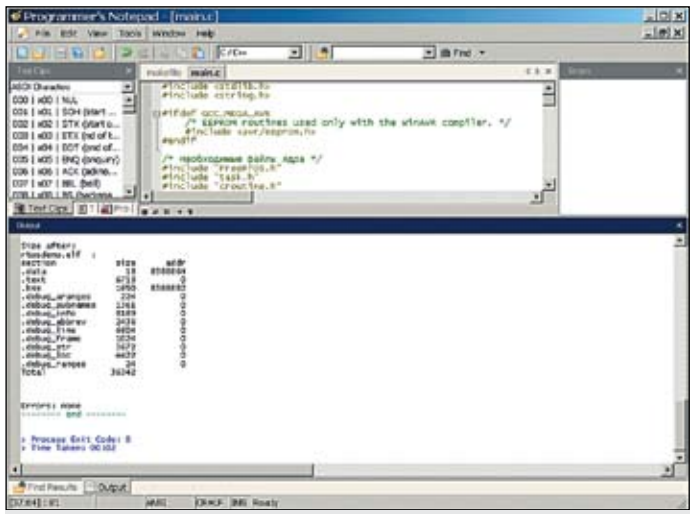

Рис. 8. Успешное завершение сборки проекта

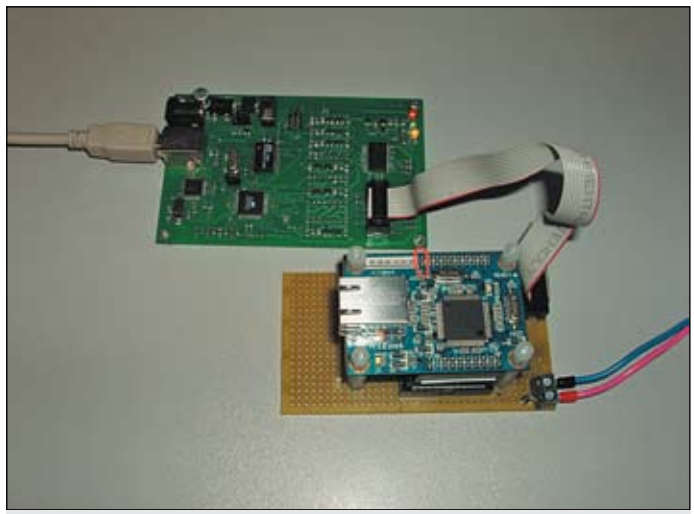

Рис. 9. Загрузка файла прошивки

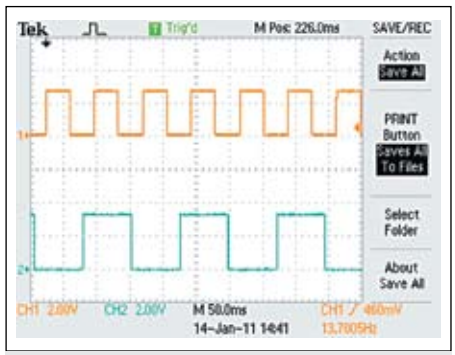

Рис. 10. Напряжение на выводах PF0 и PF1 ATmega128L (сверху вниз), полученное цифровым осциллографом

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.

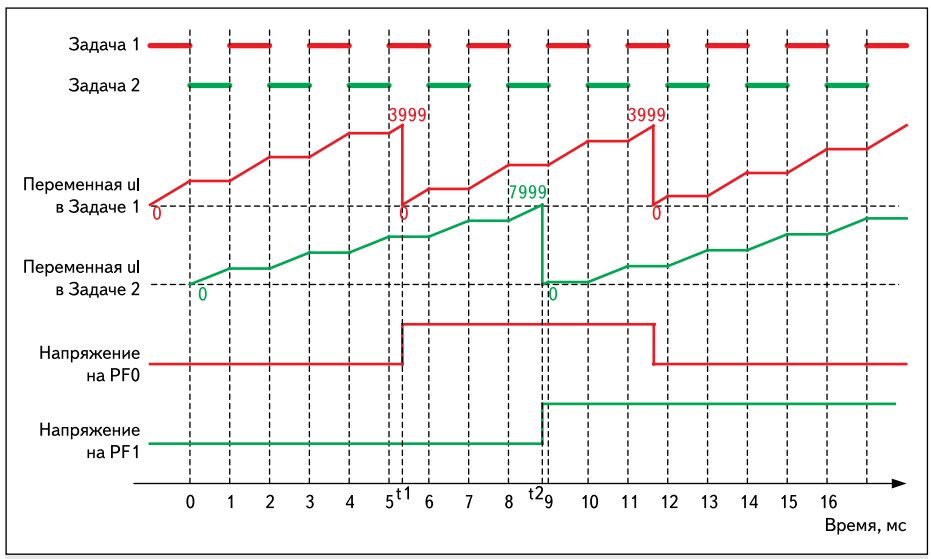

Рис. 11. Работа программы во времени

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/

Продолжение. Начало в № 2`2011

FreeRTOS — операционная система для микроконтроллеров

В предыдущих статьях [1] читатель познакомился с операционной системой реального времени (ОСРВ) для микроконтроллеров (МК) FreeRTOS. В данной статье будет продолжено изучение базовой единицы любой программы, работающей под управлением FreeRTOS, — задачи. Будет рассказано, как передать в задачу в момент ее создания произвольный параметр и как создать несколько экземпляров одной задачи. Будет показано, как блокировать задачу на определенное время и заставить ее циклически выполняться с заданной частотой. Автор использует удобную для демонстрации возможностей FreeRTOS платформу — порт FreeRTOS для x86 совместимых процессоров.

Андрей КУРНИЦ
kurnits@stim.by

Рис. 1. Включение 16-разрядного компилятора

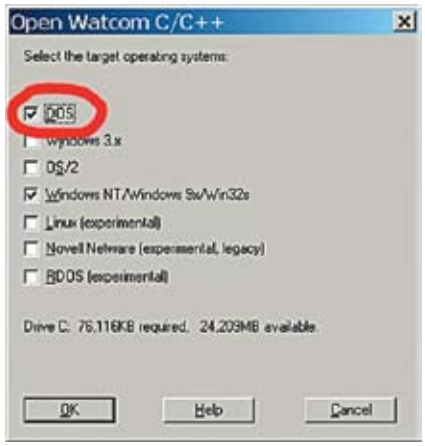

Рис. 2. Включение DOS в список целевых ОС

Подготовка к выполнению FreeRTOS на платформе x86

В предыдущей части [1] был приведен пример создания простой программы, работающей под управлением FreeRTOS. Платформой служил МК фирмы AVR ATmega128. Продолжать подробное рассмотрение и демонстрацию возможностей FreeRTOS на платформе реального МК не всегда удобно. Гораздо удобнее использовать в качестве платформы любой x86 совместимый настольный компьютер, используя соответствующий порт FreeRTOS. Все последующие примеры будут приведены для порта для x86 совместимых процессоров, работающих в реальном режиме. Мы используем бесплатный пакет Open Watcom, включающий Си-компилятор и среду разработки [2], об особенностях установки которого будет сказано ниже. Получаемые в результате компиляции и сборки исполнимые (exe) файлы могут

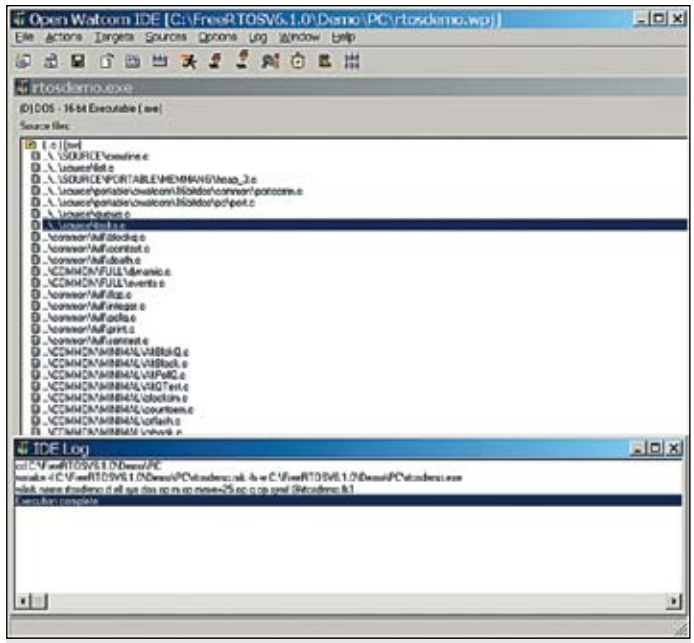

Рис. 3. Успешная сборка демонстрационного проекта в среде Open Watcom

быть выполнены из интерпретатора команд Windows (***cmd.exe***). В качестве альтернативы можно использовать бесплатный эмулятор ОС DOS под названием DOSBox, который позволит выполнять примеры не только из-под Windows, но и из-под UNIX-подобных (FreeBSD, Fedora, Gentoo Linux) и некоторых других ОС [2].

Загрузить последнюю версию пакета Open Watcom можно с официального сайта [2]. На момент написания статьи это версия 1.9. Файл для скачивания: ***open-watcom-c-win32-1.9.exe***. Во время инсталляции пакета следует включить в установку 16-разрядный компилятор для DOS и добавить DOS в список целевых ОС (рис. 1 и 2).

После установки пакета Open Watcom нужно выполнить перезагрузку рабочей станции. Далее можно проверить работу компилятора, открыв демонстрационный проект, входящий в дистрибутив FreeRTOS. Проект располагается в ***C:/FreeRTOSV6.1.0/Demo/PC/*** (в случае установки FreeRTOS на диск ***C:/***). Далее следует открыть файл проекта Open Watcom, который называется ***rtosdemo.wpj***, и выполнить сборку проекта, выбрав пункт меню ***Targets -> Make***. Сборка должна пройти без ошибок (рис. 3).

При этом в директории демонстрационного проекта появится исполнимый файл ***rtosdemo.exe***, запустив который можно наблюдать результаты работы демонстрационного проекта в окне интерпретатора команд Windows (рис. 4).

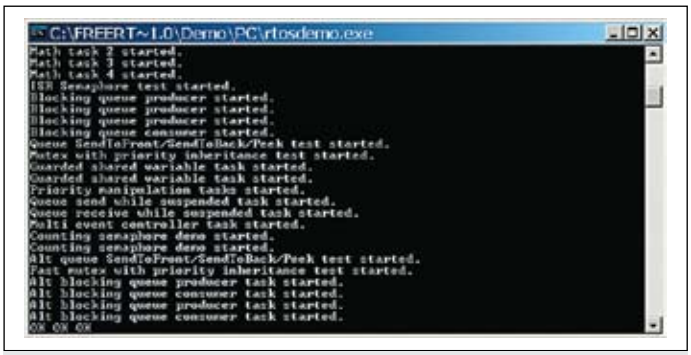

Рис. 4. Работа демонстрационного проекта в среде Windows

В демонстрационный проект включена демонстрация всех возможностей FreeRTOS. Для наших целей, чтобы продолжить изучение задач, не вникая в остальные возможности FreeRTOS, необходимо исключить из проекта все исходные и заголовочные файлы, кроме файлов ядра FreeRTOS и файла ***main.c*** (рис. 5).

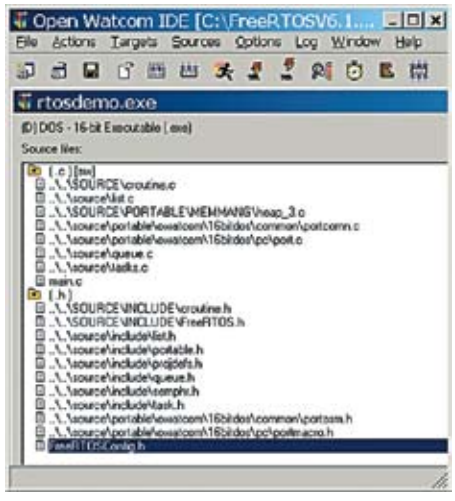

Рис. 5. Минимально необходимый набор исходных и заголовочных файлов в среде Open Watcom

Кроме этого, необходимо произвести настройку ядра, отредактировав заголовочный файл FreeRTOSConfig.h:

```
#ifndef FREERTOS_CONFIG_H
#define FREERTOS_CONFIG_H

#include <i86.h>
#include <conio.h>

#define configUSE_PREEMPTION                    1
#define configUSE_IDLE_HOOK                     0
#define configUSE_TICK_HOOK                     0
#define configTICK_RATE_HZ                      ( ( portTickType ) 1000 )
#define configMINIMAL_STACK_SIZE                ( ( unsigned short ) 256 )
                                                /* This can be made smaller if required. */
#define configTOTAL_HEAP_SIZE                   ( ( size_t ) ( 32 * 1024 ) )
#define configMAX_TASK_NAME_LEN                 ( 16 )
#define configUSE_TRACE_FACILITY                1
#define configUSE_16_BIT_TICKS                  1
#define configIDLE_SHOULD_YIELD                 1
#define configUSE_CO_ROUTINES                   0
#define configUSE_MUTEXES                       1
#define configUSE_COUNTING_SEMAPHORES           1
#define configUSE_ALTERNATIVE_API               1
#define configUSE_RECURSIVE_MUTEXES             1
#define configCHECK_FOR_STACK_OVERFLOW          0 /* Do not use this option on the PC port. */
#define configUSE_APPLICATION_TASK_TAG          1
#define configQUEUE_REGISTRY_SIZE               0

#define configMAX_PRIORITIES                    ( ( unsigned portBASE_TYPE ) 10 )
#define configMAX_CO_ROUTINE_PRIORITIES ( 2 )

/* Set the following definitions to 1 to include the API function, or zero
to exclude the API function. */

#define INCLUDE_vTaskPrioritySet                1
#define INCLUDE_uxTaskPriorityGet               1
#define INCLUDE_vTaskDelete                     1
#define INCLUDE_vTaskCleanUpResources           1
#define INCLUDE_vTaskSuspend                    1
#define INCLUDE_vTaskDelayUntil                 1
#define INCLUDE_vTaskDelay                      1
#define INCLUDE_uxTaskGetStackHighWaterMark 0 /* Do not use this option on the PC port. */

#endif /* FREERTOS_CONFIG_H */
```

Передача параметра в задачу при ее создании

На этом подготовительный этап можно считать завершенным. Как говорилось в [1], при создании задачи с помощью API-функции ***xTaskCreate()*** есть возможность передать в функцию, реализующую задачу, произвольный параметр.

Разработаем учебную программу № 1, которая будет создавать два экземпляра одной задачи. Чтобы каждый экземпляр задачи выполнял уникальное действие, передадим в качестве параметра строку символов и значение периода, которое будет сигнализировать о том, что задача выполнена. Для этого следует отредактировать файл ***main.c***:

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

/* Структура, содержащая передаваемую в задачу информацию */
typedef struct TaskParam_t {
   char           string[32];   /* строка */
   long           period;       /* период */
} TaskParam;


/* Объявление двух структур TaskParam */
TaskParam xTP1, xTP2;


/*-----------------------------------------------------------*/
/* Функция, реализующая задачу */
void vTask( void *pvParameters )
{
   volatile long ul;
   volatile TaskParam *pxTaskParam;

   /* Преобразование типа void* к типу TaskParam* */
   pxTaskParam = (TaskParam *) pvParameters;

   for( ;; )
   {
      /* Вывести на экран строку, переданную в качестве параметра при создании задачи */
      puts( (const char*)pxTaskParam->string );
```

```
        /* Задержка на некоторый период T2*/
        for( ul = 0; ul < pxTaskParam->period; ul++ )
        {
        }

    }
    vTaskDelete( NULL );
}

/*-----------------------------------------------------------*/
/* Точка входа. С функции main() начнется выполнение программы. */
short main( void )
{
    /* Заполнение полей структуры, передаваемой Задаче 1 */
    strcpy(xTP1.string, "Task 1 is running");
    xTP1.period = 10000000L;

    /* Заполнение полей структуры, передаваемой Задаче 2 */
    strcpy(xTP2.string, "Task 2 is running");
    xTP2.period = 30000000L;

    /* Создание Задачи 1. Передача ей в качестве параметра указателя на структуру xTP1 */
    xTaskCreate(  vTask,   /* Функция, реализующая задачу */
                           ( signed char * ) "Task1",
                           configMINIMAL_STACK_SIZE,
                           (void*)&xTP1, /* Передача параметра */
                           1,
                           NULL );

    /* Создание Задачи 2. Передача ей указателя на структуру xTP2 */
    xTaskCreate( vTask, ( signed char * ) "Task2", configMINIMAL_STACK_SIZE, (void*)&xTP2, 1, NULL );

    /* Запуск планировщика */
    vTaskStartScheduler();

    return 1;
}
```

Выполнив сборку проекта и запустив на выполнение полученный исполнимый файл ***rtosdemo.exe***, можно наблюдать результат работы учебной программы № 1 (рис. 6).

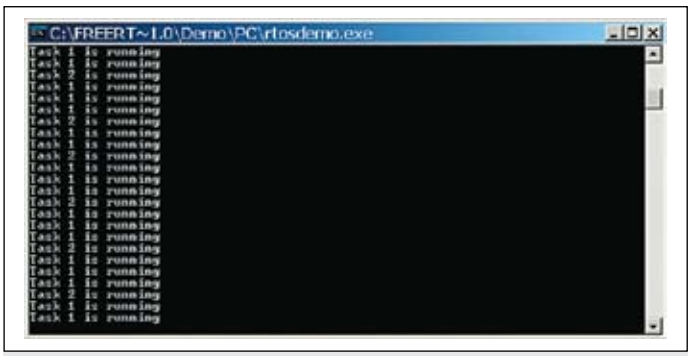

Рис. 6. Результат выполнения учебной программы № 1 в среде Windows

Задача 2 выводит сообщение о своей работе в три раза реже, чем Задача 1. Это объясняется тем, что в Задачу 2 было передано значение периода в 3 раза большее, чем в Задачу 1. Таким образом, передача различных параметров в задачи при их создании позволила добиться различной функциональности отдельных экземпляров одной задачи.

Приоритеты задач

В предыдущей статье [1] читатель познакомился с механизмом приоритетов задач. Далее будет показано, как значение приоритета влияет на выполнение задачи.

При создании задачи ей назначается приоритет. Приоритет задается с помощью параметра ***uxPriority*** функции ***xTaskCreate()***. Максимальное количество возможных приоритетов определяется макроопределением ***configMAX_PRIORITIES*** в заголовочном файле ***FreeRTOSConfig.h***. В целях экономии ОЗУ необходимо задавать наименьшее, но достаточное значение ***configMAX_PRIORITIES***. Нулевое значение приоритета соответствует наиболее низкому приоритету, значение (***configMAX_PRIORITIES-1***) — наиболее высокому (в ОС семейства Windows наоборот — приоритет 0 наивысший).

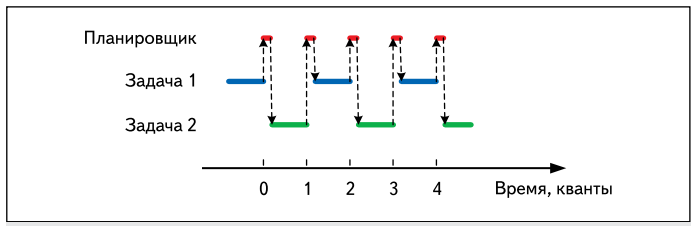

Рис. 7. Разделение процессорного времени между задачами в учебной программе № 1

Планировщик гарантирует, что среди всех задач, находящихся в состоянии готовности к выполнению, перейдет в состояние выполнения та задача, которая имеет наивысший приоритет. Если в программе созданы несколько задач с одинаковым приоритетом, то они будут выполняться в режиме разделения времени [1]. То есть задача выполняется в течение системного кванта времени, после чего планировщик переводит ее в состояние готовности и запускает следующую задачу с таким же приоритетом, и далее по кругу. Таким образом, задача выполняется за один квант времени и находится в состоянии готовности к выполнению (но не выполняется) в течение стольких квантов времени, сколько имеется готовых к выполнению задач с таким же приоритетом.

На рис. 7 показано, как задачи разделяют процессорное время в учебной программе № 1. Кроме хода выполнения двух задач, на рис. 7 показано выполнение кода планировщика каждый системный квант времени. Выполнение кода планировщика приводит к переключению на следующую задачу с одинаковым приоритетом.

Модифицируем учебную программу № 1 так, чтобы задачам назначался разный приоритет. Пусть Задача 2 получит приоритет, равный 2, а приоритет Задачи 1 останется прежним — равным 1. Для этого следует отредактировать вызов API-функции ***xTaskCreate()*** для создания Задачи 2:

```
…
xTaskCreate( vTask, ( signed char * ) "Task2", configMINIMAL_STACK_SIZE, (void*)&xTP2, 2, NULL );
…
```

Выполнив сборку модифицированной учебной программы и запустив ее на выполнение, можно наблюдать ситуацию, когда все время будет выполняться Задача 2, а Задача 1 никогда не получит управление (рис. 8).

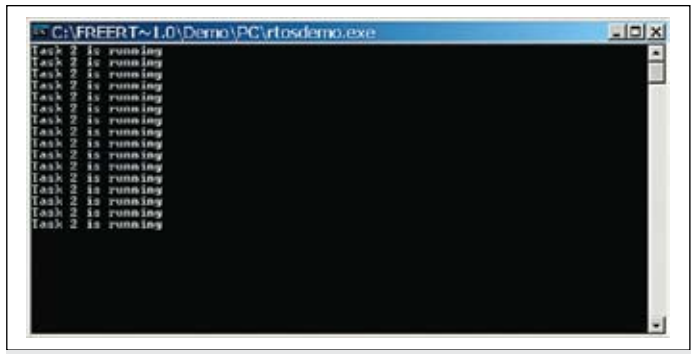

Рис. 8. Результат работы учебной программы в случае назначения Задаче 2 более высокого приоритета

Задача 2, как и Задача 1, все время находится в состоянии готовности к выполнению. За счет того, что Задача 2 имеет приоритет выше, чем Задача 1, каждый квант времени планировщик будет отдавать управление именно ей, а Задача 1 никогда не получит процессорного времени (рис. 9).

Этот пример показывает необходимость пользоваться приоритетами осмотрительно, так как никакого алгоритма старения в планировщике не предусмотрено (как в ОС общего назначения). Поэтому

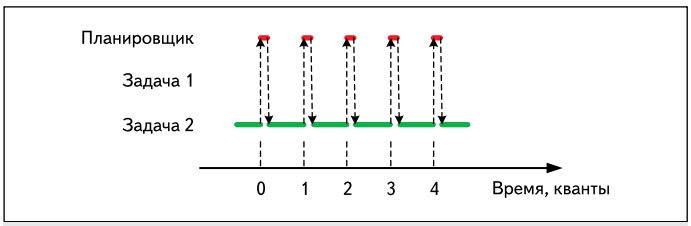

Рис. 9. Разделение времени между задачами, когда Задача 2 имеет более высокий приоритет, чем Задача 1

возможна ситуация «зависания» задачи с низким приоритетом, которая никогда не выполнится. Программисту необходимо тщательно проектировать прикладные программы и благоразумно задавать уровни приоритетов, чтобы избежать такой ситуации. Далее будет показано, как избежать «зависания» низкоприоритетных задач, используя механизм событий для управления ходом их выполнения.

Следует отметить, что FreeRTOS позволяет динамически менять приоритет задачи во время выполнения программы. Для получения и задания приоритета задачи во время выполнения служат API-функции ***uxTaskPriorityGet()*** и ***vTaskPrioritySet()*** соответственно.

Подсистема времени FreeRTOS

Подробнее остановимся на системном кванте времени. Планировщик получает управление каждый квант времени, это происходит по прерыванию от таймера. Продолжительность системного кванта определяется периодом возникновения прерываний от таймера и задается в файле ***FreeRTOSConfig.h*** макроопределением ***configTICK_RATE_HZ***.

configTICK_RATE_HZ определяет частоту отсчета системных квантов в герцах, например значение ***configTICK_RATE_HZ***, равное 100 (Гц), определяет продолжительность системного кванта, равную 10 мс. Следует отметить, что в большинстве демонстрационных проектов продолжительность системного кванта устанавливается равной 1 мс (***configTICK_RATE_HZ*** = 1000).

Все API-функции, связанные с измерением временных интервалов, в качестве единицы измерения времени используют системный квант. Используя макроопределение ***portTICK_RATE_MS***, можно получить продолжительность системного кванта в миллисекундах. Но для задания длительности кванта нужно использовать макроопределение ***configTICK_RATE_HZ***.

Следует также упомянуть о счетчике квантов — это системная переменная типа portTickType, которая увеличивается на единицу по прошествии одного кванта времени и используется ядром FreeRTOS для измерения временных интервалов. Значение счетчика квантов начинает увеличиваться после запуска планировщика, то есть после выполнения функции ***vTaskStartScheduler()***. Текущее значение счетчика квантов может быть получено с помощью API-функции ***xTaskGetTickCount()***.

События как способ управления выполнением задач

В учебных программах, приведенных выше, задачи были реализованы так, что они постоянно нуждались в процессорном времени. Даже когда задача ничего не выводила на экран, она занималась отсчетом времени с помощью пустого цикла ***for***.

Такая реализация задачи целесообразна только при назначении задаче самого низкого приоритета. В противном случае наличие такой постоянно готовой к выполнению задачи с довольно высоким приоритетом приведет к тому, что другие задачи, имеющие более низкий приоритет, никогда не будут выполняться.

Гораздо эффективнее управлять выполнением задач с помощью событий. Управляемая событием задача выполняется только после того, как некоторое событие произошло. Если событие не произошло и задача ожидает его наступления, то она НЕ находится в состоянии ГОТОВНОСТИ к выполнению, а следовательно, не может быть выполнена планировщиком. Планировщик распределяет процессорное время только между задачами, ГОТОВЫМИ к выполнению. Таким образом, если высокоприоритетная задача ожидает наступления некоторого события, то есть не находится в состоянии готовности к выполнению, то планировщик отдаст управление готовой к выполнению более низкоприоритетной задаче.

Таким образом, применение событий для управления ходом выполнения задач позволяет создавать программы с множеством различных приоритетов задач, и программист может не опасаться того, что высокоприоритетная задача «заберет» себе все процессорное время.

Блокированное состояние задачи

Если задача ожидает наступления события, то она находится в блокированном состоянии (рис. 5 в КиТ № 3`2011, стр. 111). Во FreeRTOS существуют два вида событий:

1. Временное событие — это событие, связанное с истечением временного промежутка или наступлением определенного момента абсолютного времени. Например, задача может войти в блокированное состояние, пока не пройдет 10 мс.
2. Событие синхронизации (внешнее по отношению к задаче) — это событие, которое генерируется в другой задаче или в теле обработчика прерывания МК. Например, задача блокирована, когда ожидает появления данных в очереди. Данные в очередь поступают от другой задачи.

События синхронизации могут быть связаны с множеством объектов ядра, такими как очереди, двоичные и счетные семафоры, рекурсивные семафоры и мьютексы, которые будут описаны в дальнейших публикациях.

Во FreeRTOS есть возможность заблокировать задачу, заставив ее ожидать события синхронизации, но определить при этом тайм-аут ожидания. То есть выход задачи из блокированного состояния возможен как при наступлении события синхронизации, так и по прошествии времени тайм-аута, если событие синхронизации так и не произошло. Например, задача ожидает появления данных из очереди. Тайм-аут при этом установлен равным 10 мс. В этом случае выход задачи из блокированного состояния возможен при выполнении двух условий:

- Данные в очередь поступили.
- Данные не поступили, но вышло время тайм-аута, равное 10 мс.

Реализация задержек с помощью API-функции vTaskDelay()

Вернемся к рассмотрению учебной программы № 1. Задачи в этой программе выполняли полезное действие (в нашем случае — вывод текстовой строки на экран), после чего ожидали определенный промежуток времени, то есть выполняли задержку на какое-то время. Реализация задержки в виде пустого цикла не эффективна. Один из основных недостатков мы продемонстрировали, когда задачам был назначен разный приоритет. А именно, когда высокоприоритетная задача все время остается в состоянии готовности к выполнению (не переходит ни в блокированное, ни в приостановленное состояние), она поглощает все процессорное время, и низкоприоритетные задачи никогда не выполняются.

Для корректной реализации задержек средствами FreeRTOS следует применять API-функцию ***vTaskDelay()***, которая переводит задачу, вызывающую эту функцию, в блокированное состояние на заданное количество квантов времени. Ее прототип:

```
void vTaskDelay( portTickType xTicksToDelay );
```

Единственным аргументом является ***xTicksToDelay***, который непосредственно задает количество квантов времени задержки.

Например, пусть задача вызвала функцию ***xTicksToDelay(100)*** в момент времени, когда счетчик квантов был равен 5000. Задача сразу же блокируется, планировщик отдаст управление другой задаче, а вызывающая задача вернется в состояние готовности к выполнению, только когда счетчик квантов достигнет значения 5100. В течение времени, пока счетчик квантов будет увеличиваться от 5000 до 5100, планировщик будет выполнять другие задачи, в том числе задачи с более низким приоритетом.

Следует отметить, что программисту нет необходимости отслеживать переполнение счетчика квантов времени. API-функции, связанные с отсчетом времени (в том числе и ***vTaskDelay()***), берут эту обязанность на себя.

Рассмотрим учебную программу № 2, которая выполняет те же функции, что и программа № 1, но для создания задержек в ней используется API-функция ***vTaskDelay()***. Кроме того, задаче при ее создании передается не абстрактное значение периода, а значение периода в миллисекундах:

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

/* Структура, содержащая передаваемую в задачу информацию */
typedef struct TaskParam_t {
   char string[32]; /* строка */
   long period; /* период, миллисекунды*/
} TaskParam;

/* Объявление двух структур TaskParam */
TaskParam xTP1, xTP2;

/*-----------------------------------------------------------*/
/* Функция, реализующая задачу */
void vTask( void *pvParameters )
{
   volatile TaskParam *pxTaskParam;

   /* Преобразование типа void* к типу TaskParam* */
   pxTaskParam = (TaskParam *) pvParameters;

   for( ;; )
   {
      /* Вывести на экран строку, переданную в качестве параметра
при создании задачи */
      puts( (const char*)pxTaskParam->string );
      /* Задержка на время, заданное в миллисекундах */
      /* pxTaskParam->period задан в миллисекундах /*
      /* Разделив его на кол-во мс в кванте, получим кол-во квантов */
      vTaskDelay(pxTaskParam->period / portTICK_RATE_MS);
   }
   vTaskDelete( NULL );
}

/*-----------------------------------------------------------*/
/* Точка входа. С функции main() начнется выполнение программы. */
short main( void )
{
   /* Заполнение полей структуры, передаваемой Задаче 1 */
   strcpy(xTP1.string, "Task 1 is running");
   xTP1.period = 1000L;       /* 1000 мс */

   /* Заполнение полей структуры, передаваемой Задаче 2 */
   strcpy(xTP2.string, "Task 2 is running");
   xTP2.period = 3000L;       /* 3000 мс */

   /* Создание Задачи 1 с приоритетом 1. Передача ей в качестве
параметра указателя на структуру xTP1 */
   xTaskCreate( vTask, ( signed char * ) "Task1", configMINIMAL_
STACK_SIZE, (void*)&xTP1, 1, NULL );

   /* Создание Задачи 2 с приоритетом 2. Передача ей указателя
на структуру xTP2 */
   xTaskCreate( vTask, ( signed char * ) "Task2", configMINIMAL_
STACK_SIZE, (void*)&xTP2, 2, NULL );

   /* Запуск планировщика */
   vTaskStartScheduler();

   return 1;
}
```

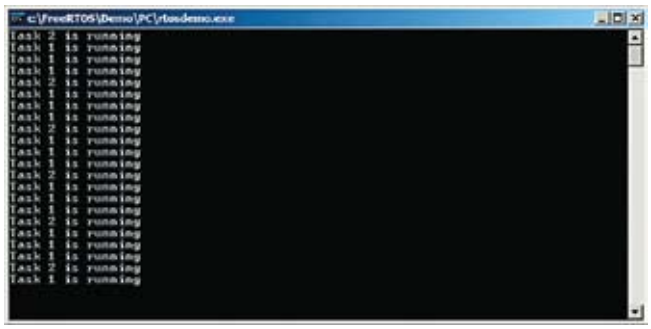

Рис. 10. Результат работы учебной программы № 2

По результатам работы учебной программы № 2 (рис. 10) видно, что процессорное время теперь получает как высокоприоритетная задача 2, так и низкоприоритетная задача 1.

Выполнение задач в учебной программе № 2 приведено на рис. 11. Выполнение кода планировщика в целях упрощения рисунка не приводится. Бóльшую часть времени процессор бездействует, а следовательно, теперь задачи очень экономно расходуют процессорное время.

В момент времени (1) происходит запуск планировщика. На этот момент Задача 1 и Задача 2 находятся в состоянии готовности к выполнению, однако приоритет Задачи 2 выше, поэтому именно ей планировщик передает управление. Задача 2 выполняет полезную работу (выводит строку ***"Task 2 is running"***) (рис. 10), после чего выполняет API-функцию ***vTaskDelay()***, в результате чего Задача 2 переходит в блокированное состояние. После вызова функции ***vTaskDelay()*** выполняемая в данный момент Задача 2 перешла в блокированное состояние и не нуждается в процессорном времени, поэтому для того чтобы занять процессор другой задачей, функция ***vTaskDelay()*** вызывает планировщик. Теперь в списке готовых к выполнению задач осталась только Задача 1, которой планировщик и отдает управление (момент времени (2)). Задача 1 выполняет свою полезную работу: также вызывает API-функцию ***vTaskDelay()*** и переходит в блокированное состояние (момент времени (3)). В этот момент нет ни одной задачи, готовой к выполнению, поэтому планировщик вызывает системную задачу, которая не выполняет никакой полезной работы, — задачу Бездействие (4). Подробнее о задаче Бездействие будет сказано ниже.

На протяжении времени, когда Задача 1 и Задача 2 находятся в блокированном состоянии, кроме выполнения задачи Бездействие, ядро FreeRTOS отсчитывает кванты времени, прошедшие с моментов вызовов API-функции ***vTaskDelay()***. Как только ядро отсчитает 1000 квантов (1000 мс), оно переведет Задачу 1 из блокированного в состояние готовности к выполнению (момент времени (5)). Планировщик отдаст ей управление, она выполнит полезную работу и снова перейдет в блокированное состояние на время 1000 мс и т. д. Задача 2 будет находиться в блокированном состоянии на протяжении 3000 мс. В момент времени (7) из блокированного состояния в состояние готовности к выполнению перейдут обе задачи, однако планировщик запустит (переведет в состояние выполнения) Задачу 2, так как приоритет у нее выше.

API-функция vTaskDelayUntil ()

API-функция ***vTaskDelayUntil()*** служит для тех же целей, что и ***vTaskDelay()***, — для перевода задачи в блокированное состояние на заданное время. Однако она имеет некоторые особенности, позволяющие с меньшими усилиями реализовать циклическое выполнение кода задачи с точно заданным периодом.

Часто перед программистом стоит задача циклического выполнения какого-либо

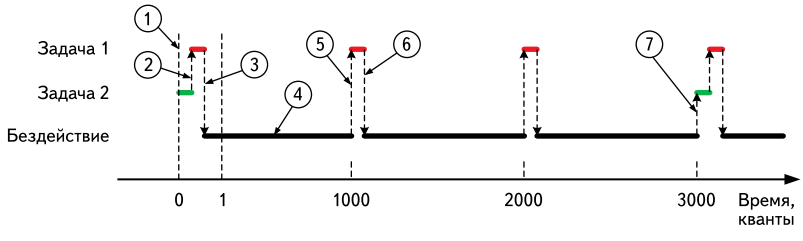

Рис. 11. Разделение процессорного времени между задачами в учебной программе № 2

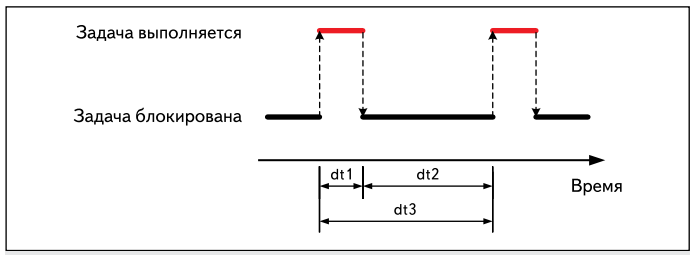

Рис. 12. Ход выполнения циклической задачи. Задержка реализована API-функцией vTaskDelay()

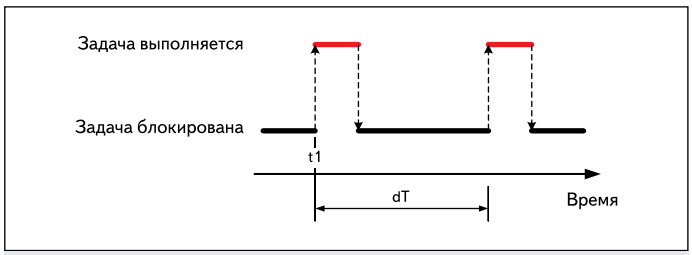

Рис. 13. Ход выполнения циклической задачи. Задержка реализована API-функцией vTaskDelayUntil()

действия с четко фиксированной частотой и, следовательно, периодом. API-функция ***vTaskDelay()*** переводит задачу в блокированное состояние на промежуток времени, который отсчитывается от момента вызова ***vTaskDelay()***. В случае реализации циклически повторяющегося действия период его выполнения *dt*3 будет складываться из времени его выполнения *dt*1 и задержки *dt*2, создаваемой функцией ***vTaskDelay()*** (рис. 12).

Если стоит цель обеспечить циклическое выполнение с точно заданным периодом *dt*3, то необходимо знать время выполнения тела задачи *dt*1, чтобы скорректировать величину задержки *dt*2. Это создает дополнительные сложности.

Для таких целей предназначена API-функция ***vTaskDelayUntil()***. Программист в качестве ее параметра задает период *dT*, который отсчитывается с момента *t*1 — момента выхода задачи из блокированного состояния (рис. 13).

Прототип функции ***vTaskDelayUntil()***:

```
void vTaskDelayUntil( portTickType * pxPreviousWakeTime, portTickType xTimeIncrement );
```

Функции ***vTaskDelayUntil()*** передаются следующие аргументы:

1. ***pxPreviousWakeTime*** — указатель на переменную, в которой хранится значение счетчика квантов в момент последнего выхода задачи из блокированного состояния (момент времени *t*1 на рис. 13). Этот момент используется как отправная точка для отсчета времени, на которое задача переходит в блокированное состояние. Переменная, на которую ссылается указатель ***pxPreviousWakeTime***, автоматически обновляется функцией ***vTaskDelayUntil()***, поэтому при типичном использовании эта переменная не должна модифицироваться в теле задачи. Исключение составляет начальная инициализация, как показано в примере ниже.
2. ***xTimeIncrement*** — непосредственно задает период выполнения задачи. Задается в квантах; для задания в миллисекундах может использоваться макроопределение ***portTICK_RATE_MS***.

Типичное применение API-функции ***vTaskDelayUntil()*** в теле функции, реализующей задачу:

```
/* Функция задачи, которая будет циклически выполняться с жестко заданным периодом в 50 мс */
void vTaskFunction( void *pvParameters )
{
   /* Переменная, которая будет хранить значение счетчика квантов
   в момент выхода задачи из блокированного состояния */
   portTickType xLastWakeTime;
   /* Переменная xLastWakeTime нуждается в инициализации текущим значением счетчика квантов.
   Это единственный случай, когда ее значение задается явно.
   В дальнейшем ее значение будет автоматически модифицироваться API-функцией vTaskDelayUntil(). */
   xLastWakeTime = xTaskGetTickCount();
   /* Бесконечный цикл */
   for( ;; )
   {
      /* Какая-либо полезная работа */
      /* ... */

      /* Период выполнения этой задачи составит 50 мс.
      Разделив это значение на кол-во миллисекунд в 1 кванте portTICK_RATE_MS,
      получим кол-во квантов периода, что и является аргументом vTaskDelayUntil().
      Переменная xLastWakeTime автоматически модифицируется внутри vTaskDelayUntil(),
      поэтому нет необходимости делать это в явном виде. */
      vTaskDelayUntil( &xLastWakeTime, ( 50 / portTICK_RATE_MS ) );
   }
}
```

Задача Бездействие

Пока на МК подано питание и он находится не в режиме энергосбережения, МК должен выполнять какой-либо код. Поэтому хотя бы одна задача должна постоянно находиться в состоянии готовности к выполнению. Однако, как показано в учебной программе № 2, может сложиться ситуация, когда все задачи в программе могут быть заблокированы.

В этом случае МК будет выполнять задачу Бездействие (Idle task). Задача Бездействие создается автоматически при запуске планировщика API-функцией ***vTaskStartScheduler()***. Задача Бездействие постоянно находится в состоянии готовности к выполнению. Ее приоритет задается макроопределением ***tskIDLE_PRIORITY*** как самый низкий в программе (обычно 0). Это гарантирует, что задача Бездействие не будет выполняться, пока в программе есть хотя бы одна задача в состоянии готовности к выполнению. Как только появится любая готовая к выполнению задача, задача Бездействие будет вытеснена ею.

Программисту предоставляется возможность добавить свою функциональность в задачу Бездействие. Для этих целей есть возможность определить функцию-ловушку (***Idle hook function***, которая является функцией обратного вызова — ***Callback function***), реализующую функциональность задачи Бездействие (далее будем называть ее функцией задачи Бездействие). Функция задачи Бездействие отличается от функции, реализующей обычную задачу. Функция задачи Бездействие не содержит бесконечного цикла, а автоматически вызывается ядром FreeRTOS множество раз, пока выполняется задача Бездействие, то есть ее тело помещается внутрь бесконечного цикла средствами ядра.

Добавление своей функциональности в функцию задачи Бездействие окажется полезным в следующих случаях:

1. Для реализации низкоприоритетных фоновых задач.
2. Для измерения резерва МК по производительности. Во время выполнения задачи Бездействие процессор не занят полезной работой, то есть простаивает. Отношение времени простоя процессора ко всему времени выполнения программы даст представление о резерве процессора по производительности, то есть о возможности добавить дополнительные задачи в программу.
3. Для снижения энергопотребления микроконтроллерного устройства. Во многих МК есть возможность перехода в режим пониженного энергопотребления для экономии электроэнергии. Это актуально, например, в случае проектирования устройства с батарейным питанием. Выход из режима энергосбережения во многих МК возможен по прерыванию от таймера. Если настроить МК так, чтобы вход в режим пониженного энергопотребления происходил в теле функции задачи Бездействие, а выход — по прерыванию от того же таймера, что используется ядром FreeRTOS для формирования квантов времени, то это позволит значительно понизить энергопотребление устройства во время простоя процессора.

Есть некоторые ограничения на реализацию функции задачи Бездействие:

1. Задачу Бездействие нельзя пытаться перевести в блокированное или приостановленное состояние.

2. Если программа допускает использование API-функции уничтожения задачи ***vTaskDelete()***, то функция задачи Бездействие должна завершать свое выполнение в течение разумного периода времени. Это требование объясняется тем, что функция задачи Бездействие ответственна за освобождение ресурсов ядра после уничтожения задачи. Таким образом, временная задержка в теле функции задачи Бездействие приведет к такой же задержке в очистке ресурсов, связанных с уничтоженной задачей, и ресурсы ядра не будут освобождены вовремя.

Чтобы задать свою функцию задачи Бездействие, необходимо в файле настройки ядра ***FreeRTOSConfig.h*** задать макроопределение ***configUSE_IDLE_HOOK*** равным 1. В одном из файлов исходного кода должна быть определена функция задачи Бездействие, которая имеет следующий прототип:

```
void vApplicationIdleHook( void );
```

Значение ***configUSE_IDLE_HOOK***, равное 0, используется, когда не нужно добавлять дополнительную функциональность.

Создадим учебную программу № 3, демонстрирующую использование функции задачи Бездействие. В программе будет определена глобальная переменная-счетчик, задача Бездействие будет инкрементировать значение этой переменной. Также будет создана задача вывода значения переменной-счетчика на экран каждые 250 мс.

Текст учебной программы № 3:

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

/* Глобальная переменная-счетчик, которая будет увеличиваться на 1
 при каждом вызове функции задачи Бездействие */
volatile unsigned long ulIdleCycleCount = 0;

/*-----------------------------------------------------------*/
/* Функция, реализующая задачу вывода на экран значения
ulIdleCycleCount
каждые 250 мс */
void vTaskFunction( void *pvParameters )
{
    /* Бесконечный цикл */
    for( ;; )
    {
        /* Вывести на экран значение переменной ulIdleCycleCount */
        printf( "ulIdleCycleCount = %lu\n\r", ulIdleCycleCount );
        /* Задержка на 250 мс */
        vTaskDelay( 250 / portTICK_RATE_MS );
    }
    vTaskDelete( NULL );
}

/*-----------------------------------------------------------*/
/* Точка входа. С функции main() начнется выполнение программы. */
short main( void )
{
    /* Создание задачи с приоритетом 1. Параметр не передается */
    xTaskCreate( vTaskFunction, ( signed char * ) "Task",
configMINIMAL_STACK_SIZE, NULL, 1, NULL );
    /* Запуск планировщика */
    vTaskStartScheduler();

    return 1;
}

/* Функция, реализующая задачу Бездействие.
 Ее имя ОБЯЗАТЕЛЬНО должно быть vApplicationIdleHook.
 Аргументов не получает. Ничего не возвращает */
void vApplicationIdleHook( void )
{
    /* Увеличить переменную-счетчик на 1 */
    ulIdleCycleCount++;
}
```

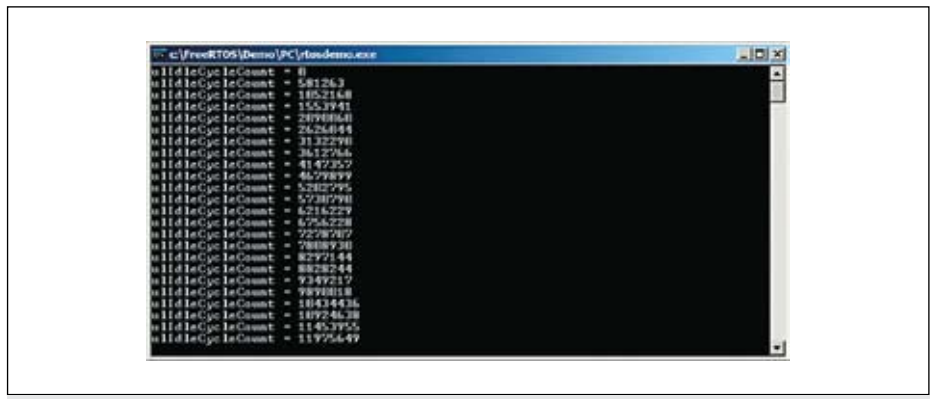

Рис. 14. Результат работы учебной программы № 3

Результат выполнения учебной программы № 3 приведен на рис. 14. Видно, что за 250 мс, пока задача вывода значения на экран пребывает в блокированном состоянии, функция задачи Бездействие «успевает выполниться» большое количество раз.

Учебная программа № 3 затрагивает еще один очень важный аспект написания программ, работающих под управлением ОСРВ, — одновременное использование одного аппаратного ресурса различными задачами. В нашем случае в качестве такого ресурса выступает глобальная переменная, доступ к которой осуществляет как задача Бездействие, так и задача отображения значения этой переменной. При совместном доступе нескольких задач к общей переменной возможна ситуация, когда выполнение одной задачи прерывается планировщиком именно в тот момент, когда задача модифицирует общую переменную, когда та еще содержит не окончательное (искаженное) значение. При этом результат работы другой задачи, которая получит управление и обратится к этой переменной, также окажется искаженным.

Однако в учебной программе № 3 задача Бездействие не может прервать операцию с общей переменной в теле задачи отображения, так как задача Бездействие будет выполняться лишь тогда, когда задача отображения завершит действия с общей переменной (вывод ее на экран функцией ***printf()***) и перейдет в блокированное состояние вызовом API-функции ***vTaskDelay()***. Одновременный совместный доступ, таким образом, исключен. Поэтому дополнительных мер для обеспечения совместного доступа к общему ресурсу в учебной программе № 3 не предпринимается.

Выводы

В статье описан способ передачи произвольного параметра в задачу при ее создании. Внимание было уделено механизму приоритетов и тому, как значение приоритета влияет на ход выполнения задачи. Рассказано о возможностях FreeRTOS для реализации задержек и периодического выполнения задачи. Изучена задача Бездействие и возможности, которые она предоставляет.

В следующих публикациях будет подробно описан процесс принудительного изменения приоритета задач в ходе их выполнения, показано, как динамически создавать и уничтожать задачи. Будет подведен итог по вытесняющей многозадачности во FreeRTOS и рассказано о возможностях кооперативной многозадачности. Далее внимание будет сфокусировано на взаимодействии и передаче информации между задачами и между прерываниями и задачами средствами FreeRTOS. ■

Литература

1. Курниц А. FreeRTOS — операционная система для микроконтроллеров // Компоненты и технологии. 2011. № 2–3.
2. http://www.openwatcom.org/index.php/Download
3. http://www.dosbox.com
4. Barry R. Using the FreeRTOS real time kernel: A Practical Guide. 2009.
5. http://www.freertos.org

Продолжение. Начало в № 2`2011

FreeRTOS — операционная система для микроконтроллеров

Андрей КУРНИЦ
kurnits@stim.by

В этой статье будет продолжено изучение FreeRTOS — операционной системы для микроконтроллеров. Здесь описан процесс принудительного изменения приоритета задач в ходе их выполнения, показано, как динамически создавать и уничтожать задачи. Рассмотрен вопрос о том, как расходуется память при создании задач. Подведен итог по вытесняющей многозадачности во FreeRTOS и рассказано о стратегии назначения приоритетов задачам под названием Rate Monotonic Scheduling. Далее мы обсудим тему кооперативной многозадачности, ее преимущества и недостатки и приведем пример программы, использующей кооперативную многозадачность во FreeRTOS. Автор уделил внимание и альтернативным схемам планирования: гибридной многозадачности и вытесняющей многозадачности без разделения времени.

Динамическое изменение приоритета

При создании задачи ей назначается определенный приоритет. Однако во FreeRTOS есть возможность динамически изменять приоритет уже созданной задачи, даже после запуска планировщика. Для динамического изменения приоритета задачи служит API-функция ***vTaskPrioritySet()***. Ее прототип:

```
void vTaskPrioritySet( xTaskHandle pxTask, unsigned portBASE_
TYPE uxNewPriority );
```

Назначение параметров:

1. ***pxTask*** — дескриптор (handle) задачи, приоритет которой необходимо изменить. Дескриптор задачи может быть получен при создании экземпляра задачи API-функцией ***xTaskCreate()*** (параметр ***pxCreatedTask*** [1, № 3]). Если необходимо изменить приоритет задачи, которая вызывает API-функцию ***vTaskPrioritySet()***, то в качестве параметра ***pxTask*** следует задать NULL.
2. ***uxNewPriority*** — новое значение приоритета, который будет присвоен задаче. При задании приоритета больше (***configMAX_PRIORITIES*** — 1) приоритет будет установлен равным (***configMAX_PRIORITIES*** — 1).

Прежде чем изменить приоритет какой-либо задачи, может оказаться полезной возможность предварительно получить значение ее приоритета. API-функция ***uxTaskPriorityGet()*** позволяет это сделать. Ее прототип:

```
unsigned portBASE_TYPE uxTaskPriorityGet( xTaskHandle pxTask );
```

Назначение параметров и возвращаемое значение:

1. ***pxTask*** — дескриптор задачи, приоритет которой необходимо получить. Если необходимо получить приоритет задачи, которая вызывает API-функцию ***uxTaskPriorityGet()***, то в качестве параметра ***pxTask*** следует задать NULL.
2. Возвращаемое значение — непосредственно значение приоритета.

Наглядно продемонстрировать использование API-функций ***vTaskPrioritySet()*** и ***uxTaskPriorityGet()*** позволяет учебная программа № 1:

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

/* Прототипы функций, которые реализуют задачи. */
void vTask1( void *pvParameters );
void vTask2( void *pvParameters );

/* Глобальная переменная для хранения приоритета Задачи 2 */
xTaskHandle xTask2Handle;

/*-----------------------------------------------------------*/

int main( void )
{
   /* Создать Задачу 1, присвоив ей приоритет 2.
   Передача параметра в задачу, как и получение дескриптора
   задачи, не используется */
   xTaskCreate( vTask1, "Task 1", 1000, NULL, 2, NULL );

   /* Создать Задачу 2 с приоритетом = 1, меньшим,
   чем у Задачи 1. Передача параметра не используется.
   Получить дескриптор создаваемой задачи в переменную
   xTask2Handle */
   xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, &xTask2Handle );

   /* Запустить планировщик. Задачи начнут выполняться.
   Причем первой будет выполнена Задача 1 */
   vTaskStartScheduler();

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

```
/* Функция Задачи 1 */
void vTask1( void *pvParameters )
{
   unsigned portBASE_TYPE uxPriority;

   /* Получить приоритет Задачи 1. Он равен 2 и не изменяется
   на протяжении всего времени
   работы учебной программы № 1 */
   uxPriority = uxTaskPriorityGet( NULL );

   for( ;; )
   {
      /* Сигнализировать о выполнении Задачи 1 */
      puts("Task1 is running" );

      /* Сделать приоритет Задачи 2 на единицу больше
      приоритета Задачи 1 (равным 3).
      Получить доступ к Задаче 2 из тела Задачи 1 позволяет
      дескриптор Задачи 2, который сохранен в глобальной
      переменной xTask2Handle*/
      puts( "To raise the Task2 priority" );
      vTaskPrioritySet( xTask2Handle, ( uxPriority + 1 ) );

      /* Теперь приоритет Задачи 2 выше. Задача 1
      продолжит свое выполнениение лишь тогда,
      когда приоритет Задачи 1 будет уменьшен. */
   }
   vTaskDelete( NULL );
}

/*-----------------------------------------------------------*/
/* Функция Задачи 2 */
void vTask2( void *pvParameters )
{
unsigned portBASE_TYPE uxPriority;

   /* Получить приоритет Задачи 2. Так как после старта
   планировщика Задача 1 имеет более высокий приоритет,
   то если Задача 2 получает управление, значит, ее приоритет
   был повышен до 3 */
   uxPriority = uxTaskPriorityGet( NULL );

   for( ;; )
   {
      /* Сигнализировать о выполнении Задачи 2 */
      puts( "Task2 is running" );

      /* Задача 2 понижает свой приоритет на 2 единицы
      (становится равен 1). Таким образом, он становится ниже
      приоритета Задачи 1, и Задача 1 получает управление */
      puts( "To lower the Task2 priority" );
      vTaskPrioritySet( NULL, ( uxPriority - 2 ) );
   }
   vTaskDelete( NULL );
}
/*-----------------------------------------------------------*/
```

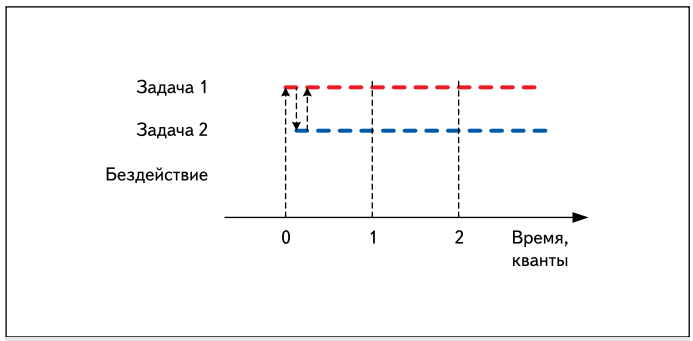

Рис. 1. Разделение процессорного времени между задачами в учебной программе № 1

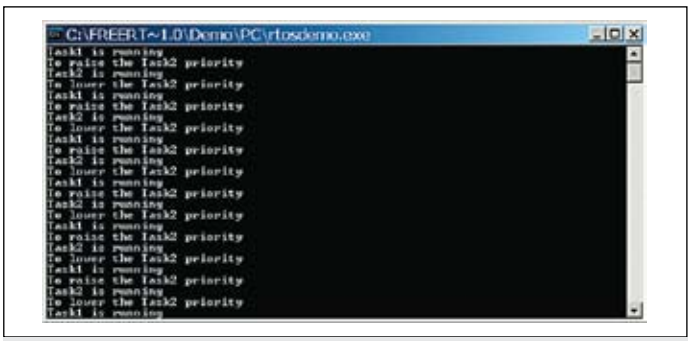

Рис. 2. Результат выполнения учебной программы № 1

В программе создана Задача 1 с приоритетом 2 и Задача 2 с приоритетом 1. Задача 1 повышает приоритет Задачи 2 так, чтобы он превысил приоритет Задачи 1. Задача 2, в свою очередь, понижает свой же приоритет так, чтобы он стал ниже приоритета Задачи 1. Задача 1, как и Задача 2, не переходит в блокированное состояние. Обе задачи поочередно получают процессорное время за счет периодического изменения приоритета Задачи 2 (он становится то ниже, то выше приоритета Задачи 1).

В момент запуска планировщика обе задачи готовы к выполнению (в учебной программе № 1 они вообще не переходят в блокированное состояние, то есть либо выполняются, либо находятся в состоянии готовности к выполнению). Управление получает Задача 1, так как ее приоритет (равен 2) больше, чем приоритет Задачи 2. После сигнализации о своей работе она вызывает API-функцию ***vTaskPrioritySet()***, вследствие чего Задача 2 получает приоритет выше, чем Задача 1 (он становится равным 3).

Вызов ***vTaskPrioritySet()*** помимо изменения приоритета приводит к тому, что управление получает планировщик, который запускает Задачу 2, так как приоритет у нее теперь выше.

Получив управление, Задача 2 сигнализирует о своем выполнении. После чего она понижает свой приоритет на две единицы, так, чтобы он стал меньше приоритета Задачи 1 (стал равен 1). После этого управление снова получает планировщик и так далее. Разделение процессорного времени в учебной программе № 1 показано на рис. 1, а результат ее выполнения — на рис. 2.

Следует отметить, что задачи сменяют друг друга с частотой, превышающей частоту системных квантов. Частота их следования зависит от быстродействия рабочей станции, и сообщения выводятся на экран с очень большой скоростью, поэтому для того, чтобы увидеть изображение на дисплее, соответствующее рис. 2, необходимо искусственно приостановить выполнение учебной программы.

Уничтожение задач

Задача может уничтожить саму себя или любую другую задачу в программе с помощью API-функции ***vTaskDelete()***. Удаленная задача физически не существует и, следовательно, никогда не выполняется. Нет возможности восстановить удаленную задачу, единственный выход — создать новую.

Ядро FreeRTOS устроено так, что внутренняя реализация задачи Бездействие отвечает за освобождение памяти, которую использовала удаленная задача. К программам, в которых происходит создание и удаление задач, предъявляется следующее требование. Если разработчик использует функцию-ловушку задачи Бездействие [1, № 4], то время выполнения этой функции должно быть меньше времени выполнения задачи Бездействие (то есть времени, пока нет ни одной задачи, готовой к выполнению).

Следует отметить, что при уничтожении задачи ядро освобождает лишь системную, не доступную прикладному программисту память, связанную с задачей. Вся память и другие ресурсы, которые программист использовал в задаче, также явно должны быть освобождены.

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

```
void vTaskDelete( xTaskHandle pxTaskToDelete );
```

Единственный параметр ***pxTaskToDelete*** — это дескриптор задачи, которую необходимо уничтожить. Если необходимо уничтожить задачу, которая вызывает API-функцию ***vTaskDelete()***, то в качестве параметра ***pxTaskToDelete*** следует задать NULL.

Учебная программа № 2 демонстрирует динамическое создание и уничтожение задач:

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

/* Прототипы функций, которые реализуют задачи. */
void vTask1( void *pvParameters );
void vTask2( void *pvParameters );

/*-----------------------------------------------------------*/

int main( void )
{
   /* Статическое создание Задачи 1 с приоритетом 1 */
   xTaskCreate( vTask1, "Task 1", 1000, NULL, 1, NULL );
   /* Запустить планировщик. Задача 1 начнет выполняться */
   vTaskStartScheduler();
   return 0;
}
/*-----------------------------------------------------------*/
/* Функция Задачи 1 */
void vTask1( void *pvParameters )
{

   for( ;; )
   {
      /* Сигнализировать о выполнении Задачи 1 */
      puts("Task1 is running" );

      /* Динамически (после старта планировщика) создать
      Задачу 2 с приоритетом 2.
      Она сразу же получит управление */
      xTaskCreate( vTask2, "Task 2", 1000, NULL, 2, NULL );

      /* Пока выполняется Задача 2 с более высоким
      приоритетом, Задача 1 не получает процессорного
      времени. Когда Задача 2 уничтожила сама себя,
      управление снова получает Задача 1 и переходит
      в блокированное состояние на 100 мс. Так что в системе
      не остается задач, готовых к выполнению, и выполняется
      задача Бездействие */
      vTaskDelay( 100 );
   }
   vTaskDelete( NULL );
}

/*-----------------------------------------------------------*/
/* Функция Задачи 2 */
void vTask2( void *pvParameters )
{
   /* Задача 2 не делает ничего, кроме сигнализации о своем
   выполнении, и сама себя уничтожает. Тело функции
   не содержит бесконечного цикла, так как в нем нет
   необходимости. Тело функции Задачи 2 выполнится 1 раз,
   после чего задача будет уничтожена. */
   puts( "Task2 is running and about to delete itself" );
   vTaskDelete( NULL );
}
/*-----------------------------------------------------------*/
```

Перед запуском планировщика создается Задача 1 с приоритетом 1. В теле Задачи 1 динамически создается Задача 2 с более высоким приоритетом. Задача 2 сразу же после создания получает управление, сигнализирует о своем выполнении и сама себя уничтожает. После чего снова управление получает Задача 1.

Следует обратить внимание на тело функции Задачи 2. В нем отсутствует бесконечный цикл, что вполне допустимо, так как функция завершается вызовом API-функции уничтожения этой задачи. Задача 2 в отличие от Задачи 1 является спорадической [5].

Разделение процессорного времени между задачами в учебной программе № 2 показано на рис. 3, а результат ее выполнения — на рис. 4.

Задача 2 существует в системе на протяжении короткого промежутка времени, пока она выполняется. Таким образом, используя

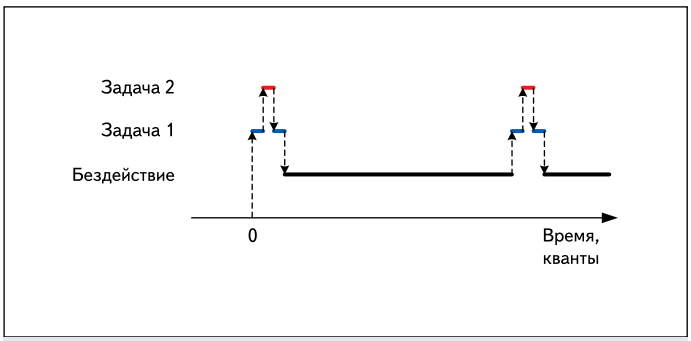

Рис. 3. Разделение процессорного времени между задачами в учебной программе № 2

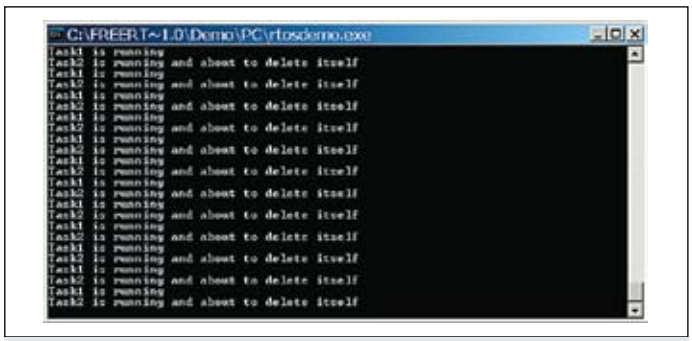

Рис. 4. Результат выполнения учебной программы № 2

динамическое создание/уничтожение задач в реальной программе, удастся достичь экономии памяти, так как память не задействуется под потребности задачи, пока полезные действия этой задачи не нужны.

Выделение памяти при создании задачи

Каждый раз при создании задачи (равно как и при создании других объектов ядра — очередей и семафоров) ядро FreeRTOS выделяет задаче блок памяти из системной кучи — области памяти, доступной для динамического размещения в ней переменных.

Блок памяти, который выделяется задаче, складывается из:

1. Стека задачи. Задается как параметр API-функции ***xTaskCreate()*** при создании задачи.
2. Блока управления задачей (Task Control Block), который представлен структурой tskTCB и содержит служебную информацию, используемую ядром. Размер структуры tskTCB зависит от:
   - настроек FreeRTOS;
   - платформы, на которой она выполняется;
   - используемого компилятора.

Размер блока памяти, который выделяется задаче, на этапе выполнения программы полностью определяется размером отводимого задаче стека, так как размер структуры tskTCB жестко задан на этапе компиляции программы и остается неизменным во время ее выполнения.

Получить точный размер структуры tskTCB для конкретных условий можно, например, добавив в текст программы следующую инструкцию:

```
printf("%d", sizeof(tskTCB));
```

И далее следует прочесть ее размер с какого-либо устройства вывода (в данном случае — с дисплея). При этом нужно учесть, что, так как структура tskTCB используется ядром в собственных целях, то доступа к этой структуре из текста прикладных исходных файлов (main.c в том числе) изначально нет. Чтобы получить доступ к структуре tskTCB, необходимо включить в исходный файл строку:

```
#include "..\\tasks.c"
```

Для учебных программ, приводимых в этой статье и ранее [1], размер структуры tskTCB составляет 70 байт.

Схемы выделения памяти

Функции динамического выделения/освобождения памяти ***malloc()*** и ***free()***, входящие в стандартную библиотеку языка Си, в большинстве случаев не могут напрямую использоваться ядром FreeRTOS, так как их использование сопряжено с рядом проблем:

- Они не всегда доступны в упрощенных компиляторах для микроконтроллеров.
- Их реализация достаточно громоздка, что приводит к дополнительному расходу памяти программ.
- Они редко являются реентерабельными [6], то есть одновременный вызов этих функций из нескольких задач может привести к непредсказуемым результатам.
- Время их выполнения не является детерминированным, то есть от вызова к вызову оно будет меняться, например, в зависимости от степени фрагментации кучи.
- Они могут усложнить конфигурацию компоновщика.

Разные приложения предъявляют различные требования к объему выделяемой памяти и временным задержкам при ее выделении. Поэтому единую схему выделения памяти невозможно применить ко всем платформам, на которые портирована FreeRTOS. Вот почему реализация алгоритма выделения памяти не входит в состав ядра, а выделена в платформенно-зависимый код (в директорию ***\Source\portable\MemMang***). Это позволяет реализовать свой собственный алгоритм выделения памяти для конкретной платформы.

Когда ядро FreeRTOS запрашивает память для своих нужд, происходит вызов API-функции ***pvPortMalloc()***, когда память освобождается — происходит вызов ***vPortFree()***. API-функции ***pvPortMalloc()*** и ***vPortFree()*** имеют такие же прототипы, как и стандартные функции ***malloc()*** и ***free()*** [7]. Реализация API-функций ***pvPortMalloc()*** и ***vPortFree()*** и представляет собой ту или иную схему выделения памяти.

Следует отметить, что API-функции ***pvPortMalloc()*** и ***vPortFree()*** можно беспрепятственно использовать и в прикладных целях, выделяя память для хранения своих переменных.

FreeRTOS поставляется с тремя стандартными схемами выделения памяти, которые содержатся соответственно в исходных файлах ***heap_1.c***, ***heap_2.c***, ***heap_3.c***. В дальнейшем будем именовать стандартные схемы выделения памяти согласно именам файлов с исходным кодом, в которых они определены. Разработчику предоставляется возможность использовать любой алгоритм выделения памяти из поставки FreeRTOS или реализовать свой собственный.

Выбор одной из стандартных схем выделения памяти осуществляется в настройках компилятора (или проекта, если используется среда разработки) добавлением к списку файлов с исходным кодом одного из файлов: ***heap_1.c***, ***heap_2.c*** или ***heap_3.c***.

Схема выделения памяти heap_1.c

Часто программа для микроконтроллера допускает только создание задач, очередей и семафоров и делает это перед запуском планировщика. В этом случае память динамически выделяется перед началом выполнения задач и никогда не освобождается. Такой подход позволяет исключить такие потенциальные проблемы при динамическом выделении памяти, как отсутствие детерминизма и фрагментация, что важно для обеспечения заданного времени реакции системы на внешнее событие.

Схема ***heap_1.c*** предоставляет очень простую реализацию API-функции ***pvPortMalloc()*** и не содержит реализации API-функции ***vPortFree()***. Поэтому такую схему следует использовать, если задачи в программе никогда не уничтожаются. Время выполнения API-функции ***pvPortMalloc()*** в этом случае является детерминированным.

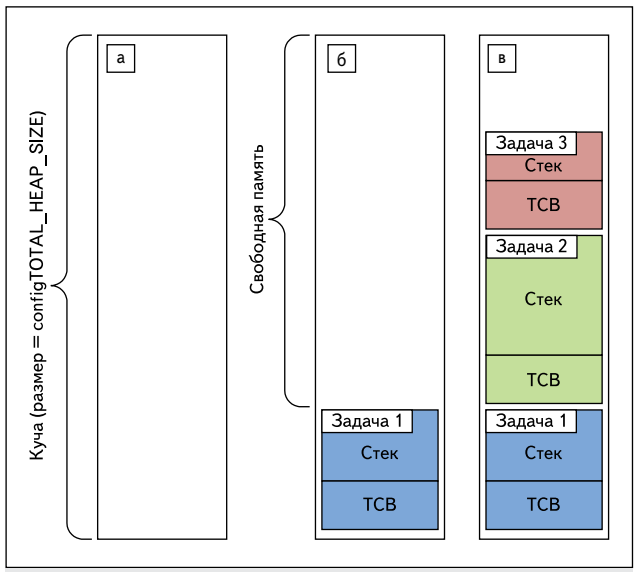

Рис. 5. Распределение памяти кучи при использовании схемы heap_1.c

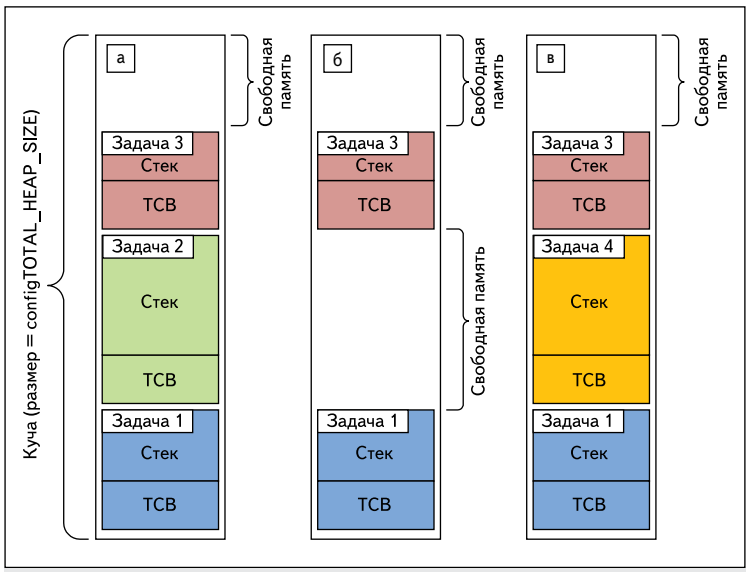

Рис. 6. Распределение памяти кучи при использовании схемы heap_2.c

Вызов ***pvPortMalloc()*** приводит к выделению блока памяти для размещения структуры tskTCB и стека задачи из кучи FreeRTOS. Выделяемые блоки памяти располагаются последовательно друг за другом (рис. 5). Куча FreeRTOS представляет собой массив байт, определенный как обычная глобальная переменная. Размер этого массива в байтах задается макроопределением ***configTOTAL_HEAP_SIZE*** в файле ***FreeRTOSConfig.h***.

Разработчик должен учесть, что объем доступной памяти для размещения переменных, связанных с решением прикладных задач, уменьшается на размер кучи FreeRTOS. Поэтому размер кучи FreeRTOS следует задавать минимальным, но достаточным для размещения всех объектов ядра. Далее будет показано, как получить объем оставшейся свободной памяти в куче FreeRTOS на этапе выполнения программы.

На рис. 5а изображена куча FreeRTOS в момент, когда ни одна задача еще не создана. На рис. 5б и в отображено размещение блоков памяти задач при их последовательном создании и, соответственно, уменьшение объема свободной памяти кучи.

Очевидно, что за счет того, что задачи не уничтожаются, эффект фрагментации памяти кучи исключен.

Схема выделения памяти heap_2.c

Как и в схеме ***heap_1.c***, память для задач выделяется из кучи FreeRTOS размером ***configTOTAL_HEAP_SIZE*** байт. Однако схема ***heap_2.c*** в отличие от ***heap_1.c*** позволяет уничтожать задачи после запуска планировщика, соответственно, она содержит реализацию API-функции ***vPortFree()***.

Так как задачи могут уничтожаться, то блоки памяти, которые они использовали, будут освобождаться, следовательно, в куче может находиться несколько отдельных участков свободной памяти (фрагментация). Для нахождения подходящего участка свободной памяти, в который с помощью API-функции ***pvPortMalloc()*** будет помещен, например, блок памяти задачи, используется алгоритм наилучших подходящих фрагментов (the best fit algorithm).

Работа алгоритма наилучших подходящих фрагментов заключается в следующем. Когда ***pvPortMalloc()*** запрашивает блок памяти заданного размера, происходит поиск свободного участка, размер которого как можно ближе к размеру запрашиваемого блока и, естественно, больше его. Например, структура кучи представляет собой 3 свободных участка памяти размером 5, 25 и 100 байт. Функция ***pvPortMalloc()*** запрашивает блок памяти 20 байт. Тогда наименьший подходящий по размеру участок памяти — участок размером 25 байт. 20 байт из этого участка будут выделены, а оставшиеся 5 байт останутся свободными.

Реализация алгоритма наилучших подходящих фрагментов в FreeRTOS не предусматривает слияния двух примыкающих друг к другу свободных участков в один большой свободный участок. Поэтому при использовании схемы ***heap_2.c*** возможна фрагментация кучи. Однако фрагментации можно не опасаться, если размер выделяемых и освобождаемых впоследствии блоков памяти не изменяется в течение выполнения программы.

Схема выделения памяти ***heap_2.c*** подходит для приложений, где создаются и уничтожаются задачи, причем размер стека при создании задач целесообразно оставлять неизменным.

На рис. 6а изображена куча FreeRTOS, блоки памяти под три задачи располагаются последовательно. На рис. 6б Задача 2 уничтожена, куча содержит два свободных участка памяти. На рис. 6в создана Задача 4 с размером стека таким же, как был у Задачи 2. В соответствии с алгоритмом наилучших подходящих фрагментов Задаче 4 выделен блок, который раньше занимала Задача 2, фрагментации кучи не произошло.

Время выполнения функций ***pvPortMalloc()*** и ***vPortFree()*** для схемы ***heap_2.c*** не является детерминированной величиной, однако их реализация значительно эффективнее стандартных функций ***malloc()*** и ***free()***.

Более подробно с существующими алгоритмами выделения памяти можно познакомиться в [8].

Схема выделения памяти heap_3.c

Схема ***heap_3.c*** использует вызовы функций выделения/освобождения памяти ***malloc()*** и ***free()*** из стандартной библиотеки языка Си. Однако с помощью останова планировщика на время выполнения этих функций достигается псевдореентерабельность (thread safe) этих функций, то есть предотвращается одновременный вызов этих функций из разных задач.

Макроопределение ***configTOTAL_HEAP_SIZE*** не влияет на размер кучи, который теперь задается настройками компоновщика.

Получение объема свободной памяти кучи

Начиная с версии V6.0.0 в FreeRTOS добавлена API-функция ***xPortGetFreeHeapSize()***, с помощью которой можно получить объем доступной для выделения свободной памяти кучи. Ее прототип:

```
size_t xPortGetFreeHeapSize( void );
```

Однако следует учесть, что API-функция ***xPortGetFreeHeapSize()*** доступна только при

использовании схем ***heap_1.c*** и ***heap_2.c***. При использовании схемы ***heap_3.c*** получение объема доступной памяти становится нетривиальной задачей.

Резюме по вытесняющей многозадачности в FreeRTOS

Подведя итог по вытесняющей многозадачности в FreeRTOS, можно выделить следующие основные принципы:

1. Каждой задаче назначается приоритет.
2. Каждая задача может находиться в одном из нескольких состояний (выполнение, готовность к выполнению, блокированное состояние, приостановленное состояние).
3. В один момент времени только одна задача может находиться в состоянии выполнения.
4. Планировщик переводит в состояние выполнения готовую к выполнению задачу с наивысшим приоритетом.
5. Задачи могут ожидать наступления события, находясь в блокированном состоянии.
6. События могут быть двух основных типов:
   - временные события;
   - события синхронизации.
7. Временные события чаще всего связаны с организацией периодического выполнения каких-либо полезных действий или с отсчетом времени тайм-аута.
8. События синхронизации чаще всего связаны с обработкой асинхронных событий внешнего мира, например, с получением информации от периферийных (по отношению к процессору) устройств.

Такая схема называется вытесняющим планированием с фиксированными приоритетами (Fixed Priority Preemptive Scheduling). Говорят, что приоритеты фиксированы, потому что планировщик самостоятельно не может изменить приоритет задачи, как это происходит при динамических алгоритмах планирования [5]. Приоритет задаче назначается в явном виде при ее создании, и так же в явном виде он может быть изменен этой же или другой задачей. Таким образом, программист целиком и полностью контролирует приоритеты задач в системе.

Стратегия назначения приоритетов задачам

Как было сказано ранее, программа, выполняющаяся под управлением FreeRTOS, представляет собой совокупность взаимодействующих задач. Чаще всего задача реализуется как какое-либо полезное действие, которое циклически повторяется с заданной частотой/периодом. Каждой задаче назначаются приоритет и частота ее циклического выполнения. Для достижения заданного времени реакции системы на внешние события разработчик должен соответствующим образом назначить приоритеты задачам и определить частоту их выполнения.

Так как FreeRTOS относится к ОСРВ с планированием с фиксированными приоритетами, то рекомендованной стратегией назначения приоритетов является использование принципа «чем меньше период выполнения задачи, тем выше у нее приоритет» (Rate Monotonic Scheduling, RMS) [4, 5].

Основная идея принципа RMS состоит в следующем. Все задачи разделяются по требуемому времени реакции на соответствующее задаче внешнее событие. Каждой задаче назначается уникальный приоритет (то есть в программе не должно быть двух задач с одинаковым приоритетом), причем приоритет тем выше, чем короче время реакции задачи на событие. Частота выполнения задачи устанавливается тем больше, чем больше ее приоритет. Таким образом, самой «ответственной» задаче назначаются наивысший приоритет и наибольшая частота выполнения.

Принцип RMS гарантирует, что система будет иметь детерминированное время реакции на внешнее событие [5]. Однако тот факт, что задачи могут изменять свой приоритет и приоритет других задач во время выполнения, и то, что не все задачи реализуются как циклически выполняющиеся, делают это утверждение в общем случае неверным, и разработчик вынужден прибегать к дополнительным мерам обеспечения заданного времени реакции.

Кооперативная многозадачность во FreeRTOS

До этого момента при изучении FreeRTOS мы использовали режим работы ядра с вытесняющей многозадачностью. Тем не менее, кроме вытесняющей, FreeRTOS поддерживает кооперативную и гибридную (смешанную) многозадачность.

Самое весомое отличие кооперативной многозадачности от вытесняющей — то, что планировщик не получает управление каждый системный квант времени. Вместо этого тело функции, реализующей задачу, должно содержать явный вызов API-функции планировщика ***taskYIELD()***.

Результатом вызова ***taskYIELD()*** может быть как переключение на другую задачу, так и отсутствие переключения, если других задач, готовых к выполнению, нет. Вызов API-функции, которая переводит задачу в блокированное состояние, также приводит к вызову планировщика.

Следует отметить, что отсчет квантов времени ядро FreeRTOS выполняет при использовании любого типа многозадачности, поэтому API-функции, связанные с отсчетом времени, корректно работают и в режиме кооперативной многозадачности. Как и для вытесняющей, в случае применения кооперативной многозадачности каждой задаче необходим собственный стек для хранения своего контекста.

Преимущества кооперативной многозадачности:

1. Меньшее потребление памяти стека при переключении контекста задачи, соответственно, более быстрое переключение контекста. С точки зрения компилятора вызов планировщика «выглядит» как вызов функции, поэтому в стеке автоматически сохраняются регистры процессора и нет необходимости их повторного сохранения в рамках сохранения контекста задачи.
2. Существенно упрощается проблема совместного доступа нескольких задач к одному аппаратному ресурсу. Например, не нужно опасаться, что несколько задач одновременно будут модифицировать одну переменную, так как операция модификации не может быть прервана планировщиком.

Недостатки:

1. Программист должен в явном виде вызывать API-функцию ***taskYIELD()*** в теле задачи, что увеличивает сложность программы.
2. Одна задача, которая по каким-либо причинам не вызвала API-функцию ***taskYIELD()***, приводит к «зависанию» всей программы.
3. Трудно гарантировать заданное время реакции системы на внешнее событие, так как оно зависит от максимального временного промежутка между вызовами ***taskYIELD()***.
4. Вызов ***taskYIELD()*** внутри циклов может замедлить выполнение программы.

Для выбора режима кооперативной многозадачности необходимо задать значение макроопределения ***configUSE_PREEMPTION*** в файле ***FreeRTOSConfig.h*** равным 0:

```
#define configUSE_PREEMPTION        0
```

Значение ***configUSE_PREEMPTION***, равное 1, дает предписание ядру FreeRTOS работать в режиме вытесняющей многозадачности.

Если включить режим кооперативной многозадачности в учебной программе № 1 [1, № 4] так, как показано выше, выполнить сборку проекта и запустить на выполнение полученный исполнимый файл ***rtosdemo.exe***, то можно наблюдать ситуацию, когда все время выполняется один экземпляр задачи, а второй никогда не получает управления (рис. 8, см. КиТ № 4`2011, стр. 98).

Это происходит из-за того, что планировщик никогда не получает управления и не может запустить на выполнение другую задачу. Теперь обязанность запуска планировщика ложится на программиста.

Если добавить в функцию, реализующую задачу, явный вызов планировщика API-функцией ***taskYIELD()***:

```
/*-----------------------------------------------------------*/
/* Функция, реализующая задачу */
void vTask( void *pvParameters )
{
   volatile long ul;
   volatile TaskParam *pxTaskParam;

   /* Преобразование типа void* к типу TaskParam* */
   pxTaskParam = (TaskParam *) pvParameters;

   for( ;; )
   {
      /* Вывести на экран строку, переданную в качестве
      параметра при создании задачи */
      puts( (const char*)pxTaskParam->string );
      /* Задержка на некоторый период T2*/
      for( ul = 0; ul < pxTaskParam->period; ul++ )
      {
      }
      /* Принудительный вызов планировщика.
      Другой экземпляр задачи получит управление
      и будет выполняться, пока не вызовет taskYIELD()
      или блокирующую API-функцию */
      taskYIELD();

   }
   vTaskDelete( NULL );
}
```

то процессорное время теперь будут получать оба экземпляра задачи. Результат выполнения программы не будет отличаться от приведенного на рис. 6 (см. КиТ № 4'2011, стр. 98). Но разделение процессорного времени между задачами будет происходить иначе (рис. 7).

На рис. 7 видно, что теперь Задача 1, как только начала выполняться, захватывает процессор на длительное время, до тех пор пока в явном виде не вызовет планировщик API-функцией ***taskYIELD()*** (момент времени N). После вызова планировщика он передает управление Задаче 2, которая тоже удерживает процессор в своем распоряжении до вызова ***taskYIELD()*** (момент времени M). Планировщик теперь не вызывается каждый квант времени, а «ждет», когда его вызовет одна из задач.

Гибридная многозадачность во FreeRTOS

Гибридная многозадачность сочетает в себе автоматический вызов планировщика каждый квант времени, а также возможность принудительного, явного вызова планировщика. Полезной гибридная многозадачность может оказаться, когда необходимо сократить время реакции системы на прерывание. В этом случае в конце тела

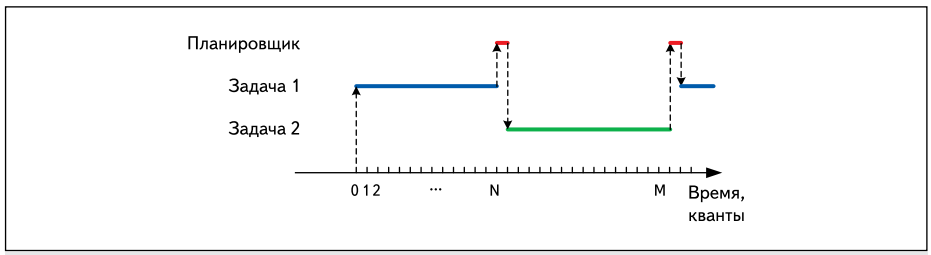

Рис. 7. Разделение процессорного времени между задачами при кооперативной многозадачности при явном вызове taskYIELD()

обработчика прерывания производят вызов планировщика, что приводит к переключению на задачу, ожидающую наступления этого прерывания.

API-функция ***portYIELD_FROM_ISR()*** служит для вызова планировщика из тела обработчика прерывания. Более подробно о ней будет рассказано позже, при изучении двоичных семафоров.

Какого-либо специального действия для включения режима гибридной многозадачности не существует. Достаточно разрешить вызов планировщика каждый квант времени (макроопределение ***configUSE_PREEMPTION*** в файле ***FreeRTOSConfig.h*** должно быть равным 1) и в явном виде вызывать планировщик в функциях, реализующих задачи, и в обработчиках прерываний с помощью API-функций ***taskYIELD()*** и ***portYIELD_FROM_ISR()*** соответственно.

Вытесняющая многозадачность без разделения времени

Ее идея заключается в том, что вызов планировщика происходит только в обработчиках прерываний. Задача выполняется до тех пор, пока не произойдет какое-либо прерывание. После чего она вытесняется задачей, ответственной за обработку внешнего события, связанного с этим прерыванием. Таким образом, задачи не сменяют друг друга по прошествии кванта времени, это происходит только по внешнему событию.

Такой тип многозадачности более эффективен в отношении производительности, чем вытесняющая многозадачность с разделением времени. Процессорное время не тратится впустую на выполнение кода планировщика каждый квант времени.

Для использования этого типа многозадачности макроопределение ***configUSE_PREEMPTION*** в файле ***FreeRTOSConfig.h*** должно быть равным 0 и каждый обработчик прерывания должен содержать явный вызов планировщика ***portYIELD_FROM_ISR()***.

Выводы

На этом изучение задачи как базовой единицы программы, работающей под управлением FreeRTOS, можно считать завершенным. Каждая задача представляет собой отдельную подпрограмму, которая работает независимо от остальных. Однако задачи не могут функционировать изолированно. Они должны обмениваться информацией и координировать свою совместную работу. Во FreeRTOS основным средством обмена информацией между задачами и средством синхронизации задач является механизм очередей.

Поэтому следующие публикации будут посвящены очередям. Подробно будет рассказано:

- как создать очередь;
- каким образом информация хранится и обрабатывается очередью;
- как передать данные в очередь;
- как получить данные из очереди;
- как задачи блокируются, ожидая возможности записать данные в очередь или получить их оттуда;
- какой эффект оказывает приоритет задач при записи и чтении данных в/из очереди.

Литература

1. Курниц А. FreeRTOS — операционная система для микроконтроллеров // Компоненты и технологии. 2011. № 2–4.
2. Barry R. Using the freertos real time kernel: A Practical Guide. 2009.
3. http://www.freertos.org
4. http://en.wikipedia.org/wiki/Rate-monotonic_scheduling
5. http://www.4stud.info/rtos/lecture3.html
6. http://ru.wikipedia.org/wiki/Реентерабельность
7. http://ru.wikipedia.org/wiki/Malloc
8. http://peguser.narod.ru/translations/files/tlsf.pdf

Продолжение. Начало в № 2`2011

FreeRTOS — операционная система для микроконтроллеров

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

Андрей КУРНИЦ
kurnits@stim.by

Необходимость использования очередей

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

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

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

Текст учебной программы № 1:

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

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

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

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

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

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

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

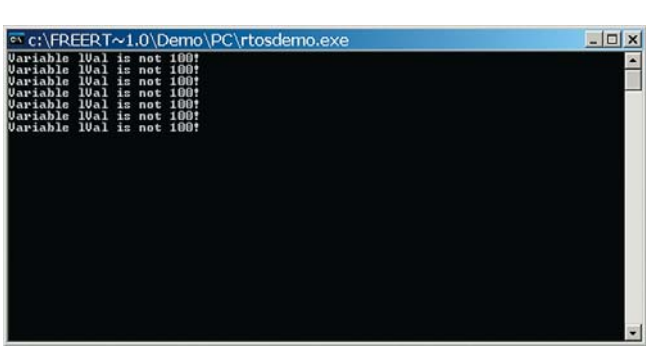

Рис. 1. Результат выполнения учебной программы № 1

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

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

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

Характеристики очередей

Хранение информации в очереди

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

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

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

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

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

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

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

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

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

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

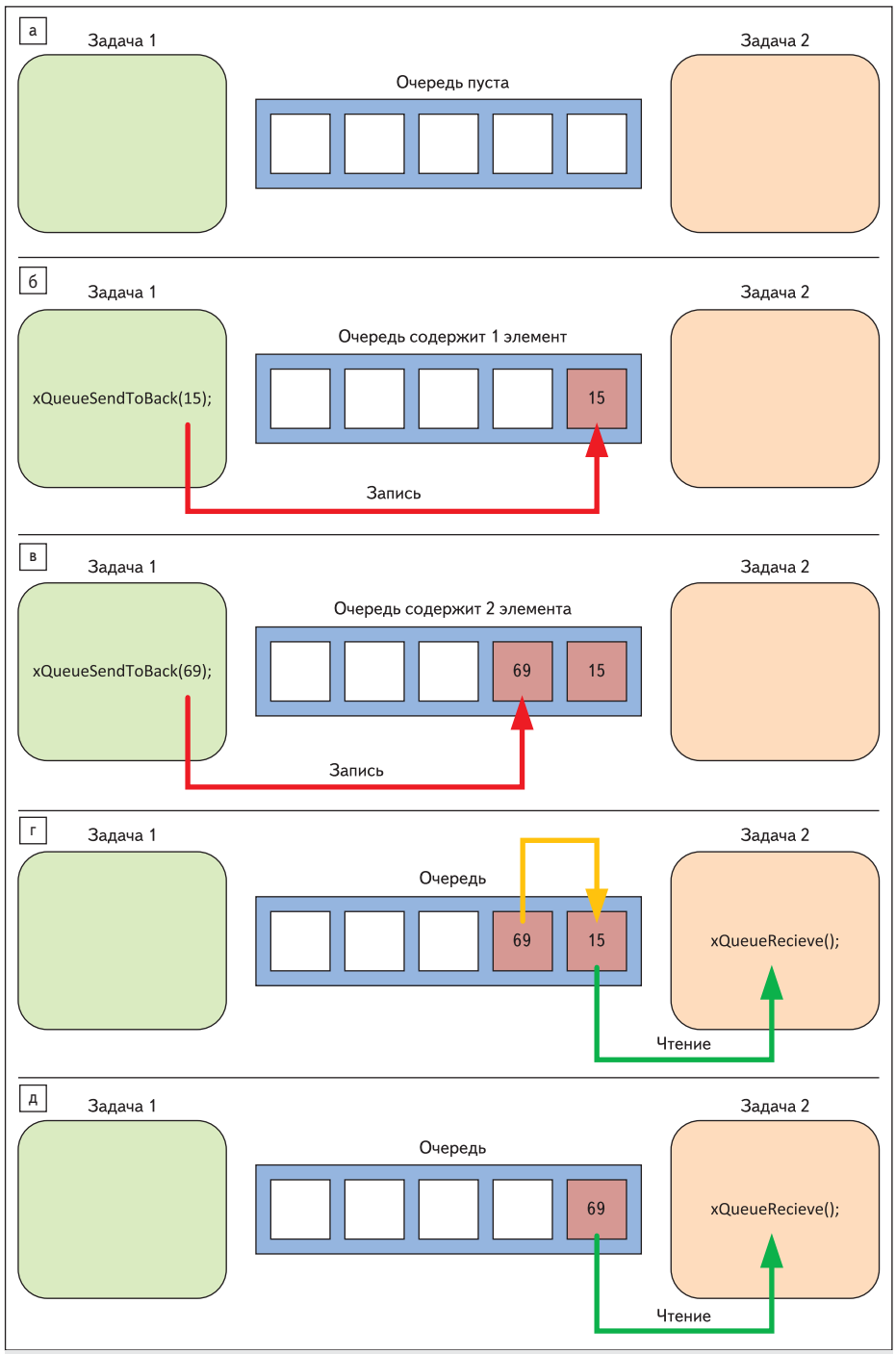

Рис. 2. Запись и чтение элементов из очереди по принципу FIFO

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

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

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

Доступ из множества задач

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

Блокировка при чтении из очереди

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

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

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

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

Блокировка при записи в очередь

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

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

Работа с очередями

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

Создание очереди

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

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

Ее параметры и возвращаемое значение:

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

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

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

```
sizeof(xQUEUE)
```

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

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

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

Запись элемента в очередь

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

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

Назначение параметров и возвращаемое значение:

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

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

Чтение элемента из очереди

Чтение элемента из очереди может быть произведено двумя способами:

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

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

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

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

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

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

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

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

Состояние очереди

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

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

Назначение параметров и возвращаемое значение:

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

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

Удаление очереди

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

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

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

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

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

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

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

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

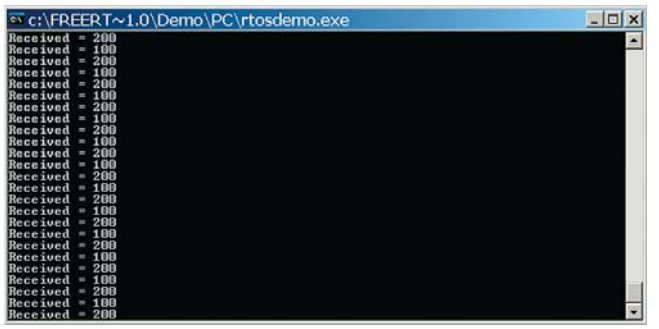

Рис. 3. Результат выполнения учебной программы № 2

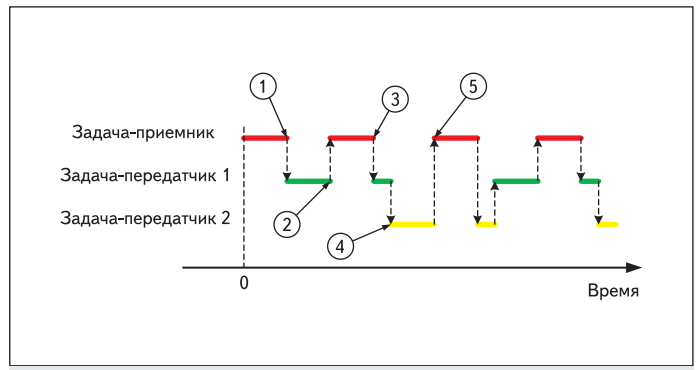

Рис. 4. Последовательность выполнения задач в учебной программе № 2

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


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

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

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

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

Использование очередей для передачи составных типов

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

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

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

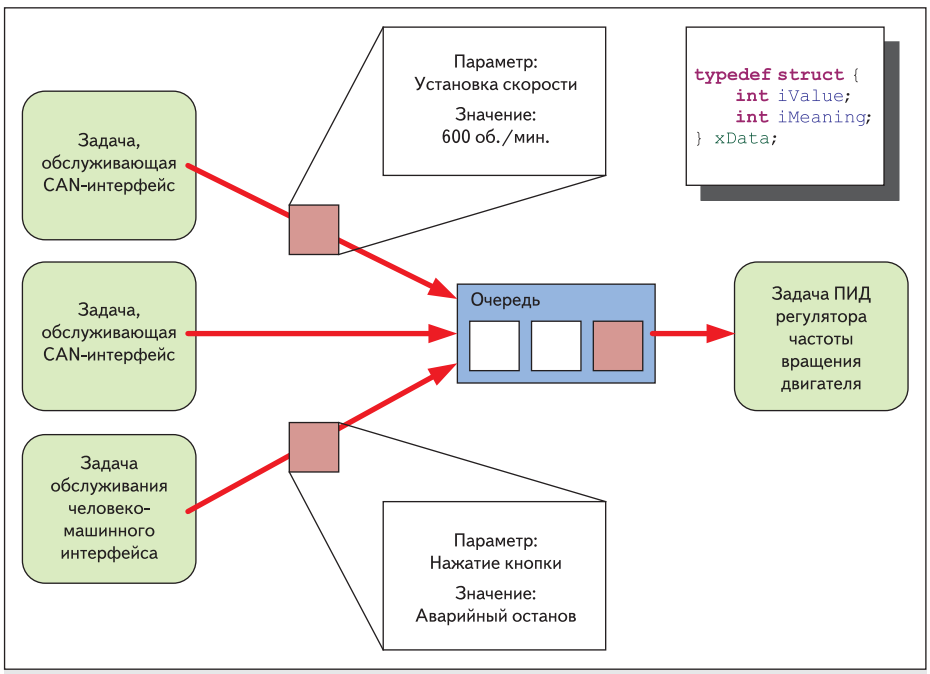

Рис. 5. Пример организации обмена информацией между задачами

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

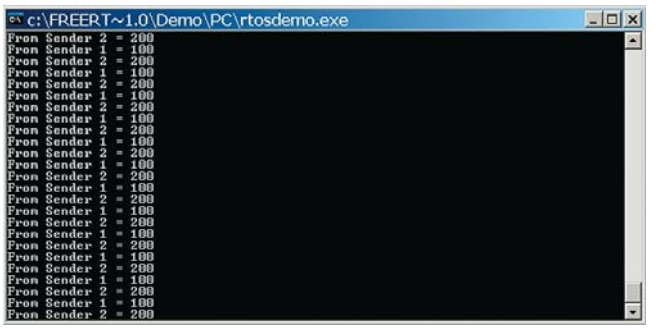

Рис. 6. Результат выполнения учебной программы № 3

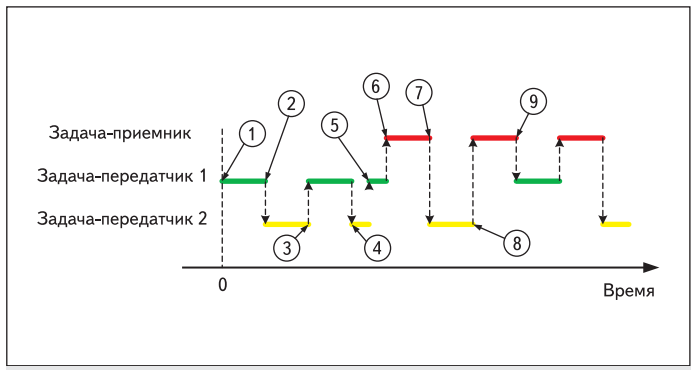

Рис. 7. Последовательность выполнения задач в учебной программе № 3

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

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

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

Использование очередей для передачи больших объемов данных

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

Преимущества такого подхода:

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

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

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

Выводы

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

Литература

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

Продолжение. Начало в № 2`2011

FreeRTOS — операционная система для микроконтроллеров

Андрей КУРНИЦ
kurnits@stim.by

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

Введение

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

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

События и прерывания

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

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

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

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

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

Отложенная обработка прерываний

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

Двоичные семафоры

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

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

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

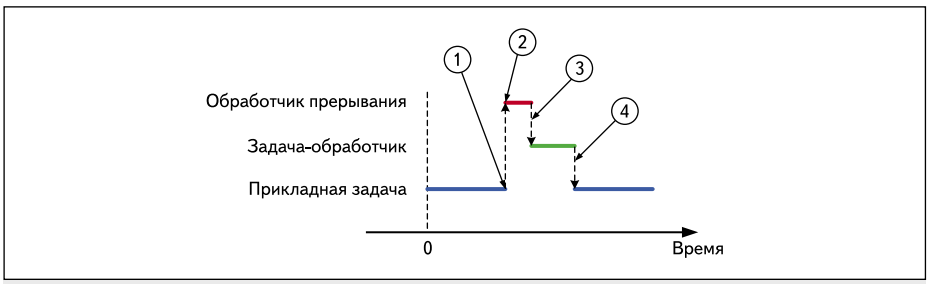

Рис. 1. Отложенная обработка прерывания с использованием двоичного семафора

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

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

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

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

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

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

Работа с двоичными семафорами

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

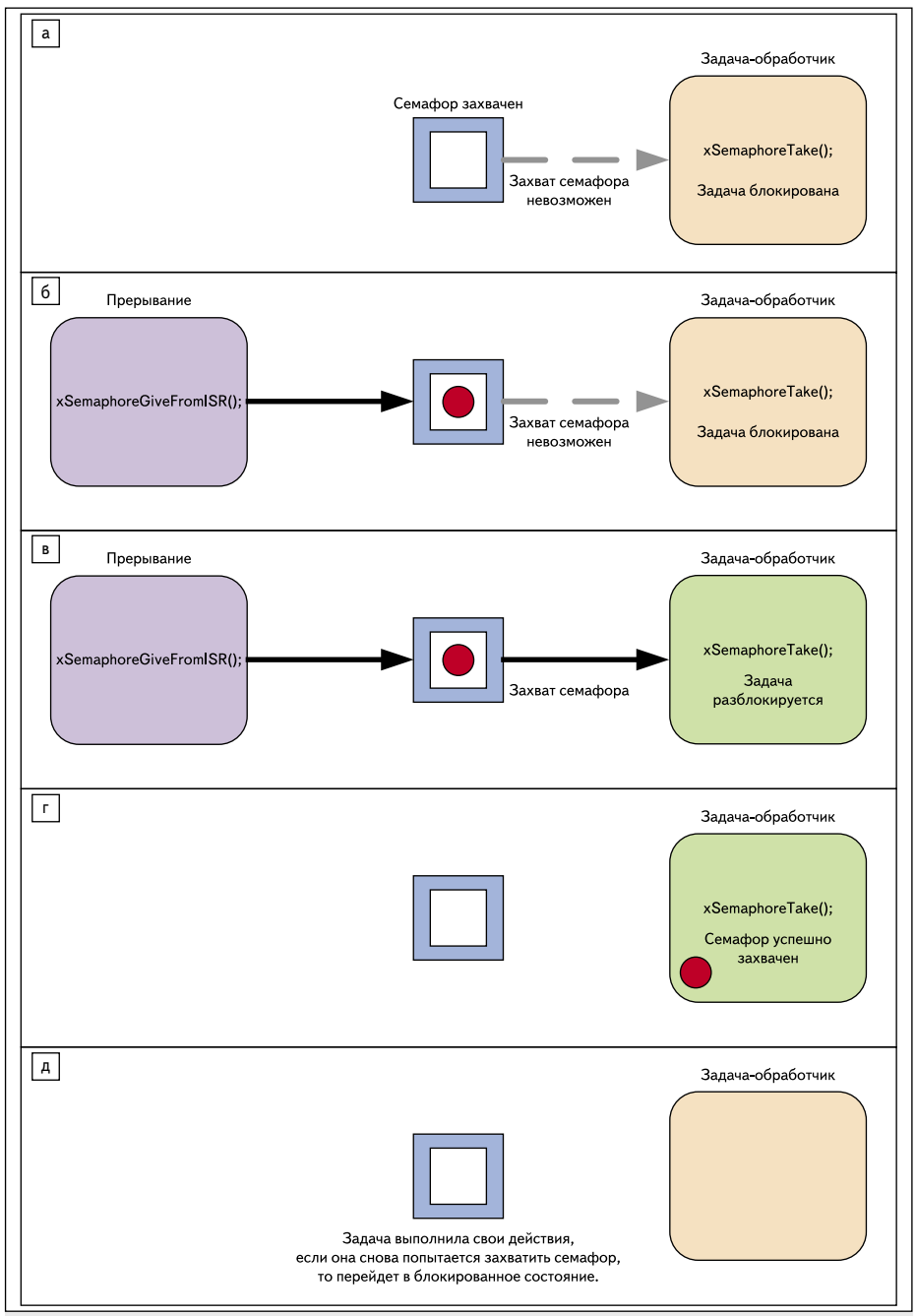

Рис. 2. Синхронизация прерывания и задачи-обработчика с помощью двоичного семафора

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

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

Создание двоичного семафора

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

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

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

Захват семафора

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

Прототип:

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

Назначение параметров и возвращаемое значение:

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

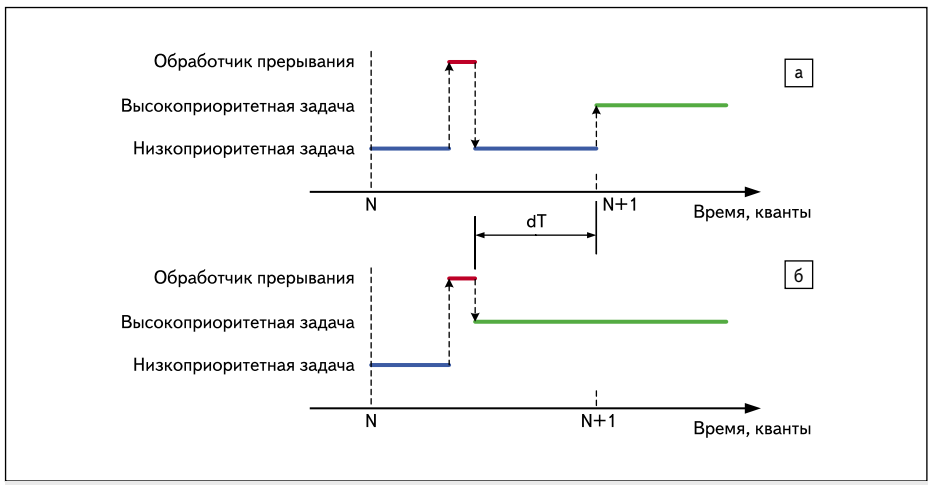

Рис. 3. Результат вызова xSemaphoreGiveFromISR():
а) без принудительного переключения контекста; б) с принудительным переключением контекста

Выдача семафора из обработчика прерывания

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

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

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

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

Назначение параметров и возвращаемое значение:

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

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

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

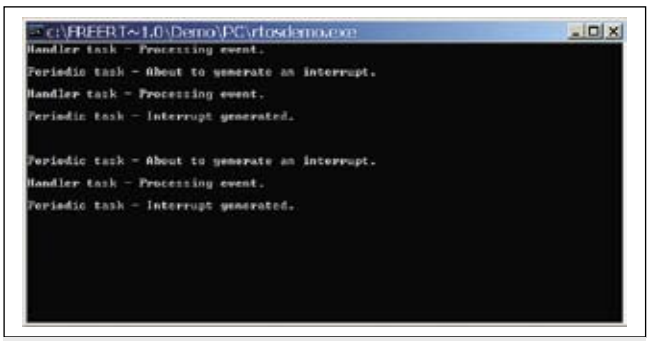

Рис. 4. Результаты выполнения учебной программы № 1

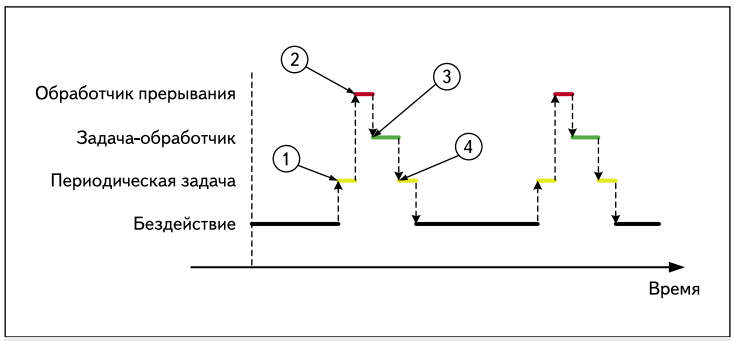

Рис. 5. Последовательность выполнения задач в учебной программе № 1

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

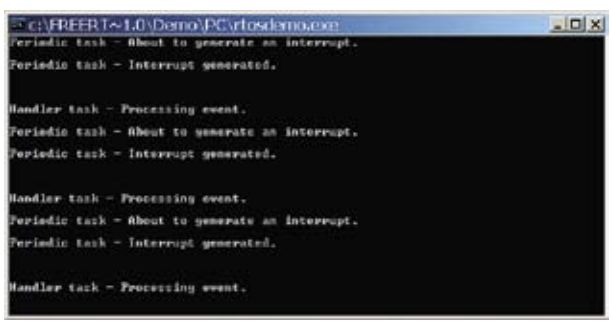

Рис. 6. Результаты выполнения учебной программы № 1 при отсутствии принудительного переключения контекста

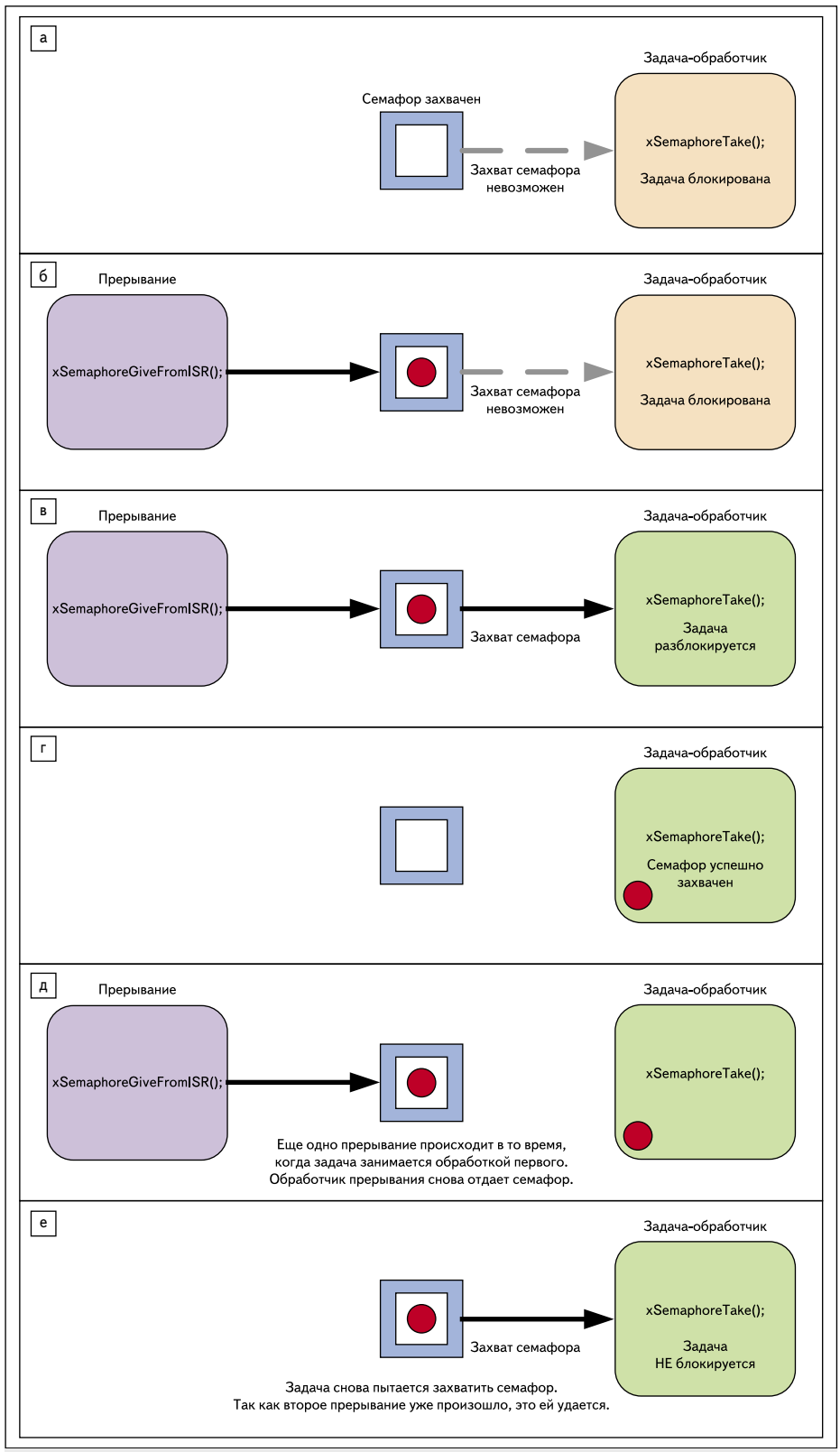

Рис. 7. «Потеря» прерывания при обработке с помощью двоичного семафора

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

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

```
portSWITCH_CONTEXT();
```

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

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

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

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

Счетные семафоры

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

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

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

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

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

Существует два основных применения счетных семафоров:

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

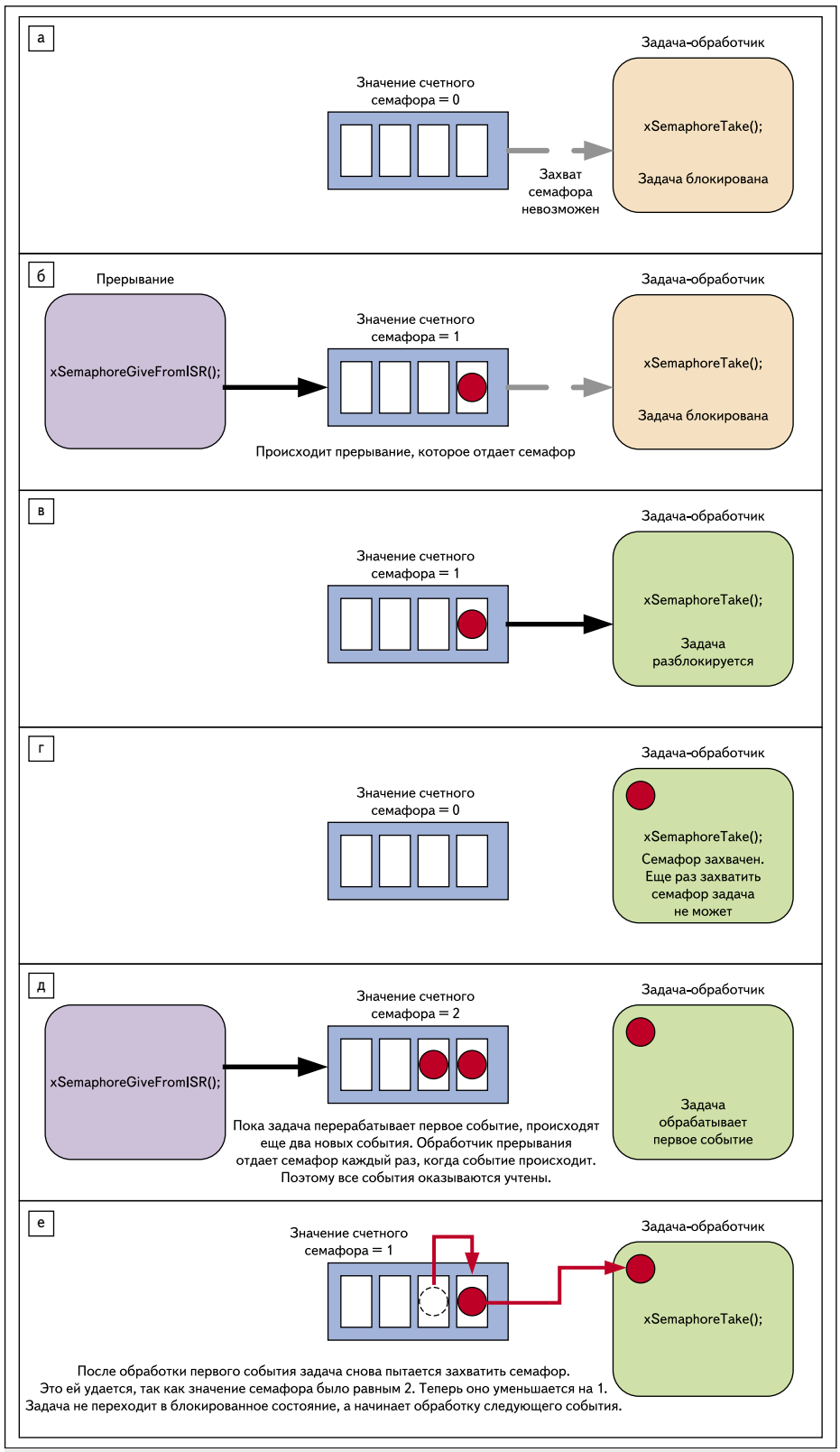

Рис. 8. Подсчет событий с помощью счетного семафора

Работа со счетными семафорами

Создание счетного семафора

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

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

Назначение параметров и возвращаемое значение:

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

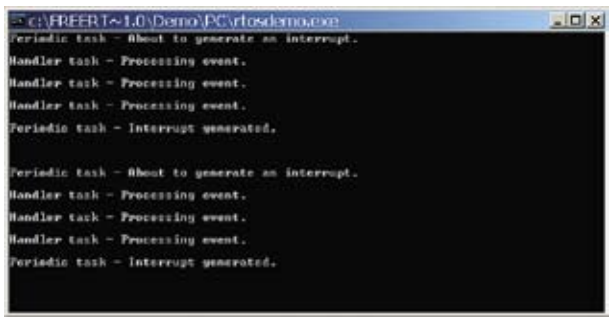

Рис. 9. Обработка быстро следующих событий

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

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

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

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

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

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

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

следует заменить функцией создания счетного семафора:

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

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

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

Использование очередей в обработчиках прерываний

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

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

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

Рассмотрим их прототипы:

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

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

Аргументы и возвращаемое значение:

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

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

Ее прототип:

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

Аргументы и возвращаемое значение:

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

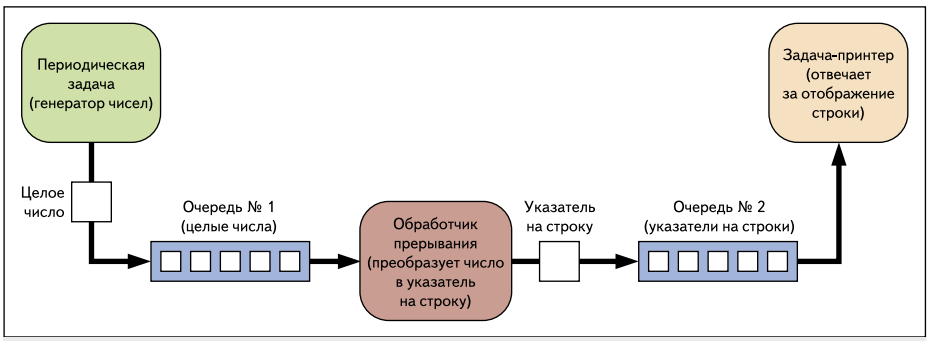

Рис. 10. Обмен данными между задачами и прерыванием в учебной программе № 2

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

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

Эффективное использование очередей

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

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

Гораздо эффективнее использовать один из следующих подходов:

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

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

Текст учебной программы № 2:

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

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

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

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

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

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

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

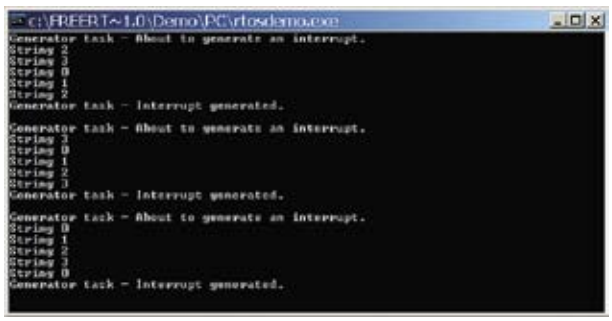

Рис. 11. Результаты выполнения учебной программы № 2

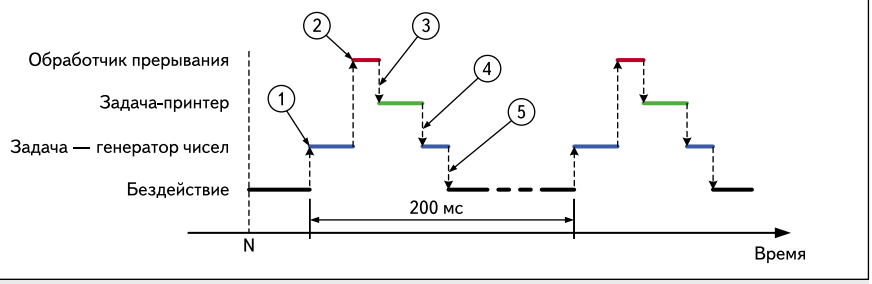

Рис. 12. Последовательность выполнения задач и прерываний в учебной программе № 2

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

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

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

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

Вложенность прерываний

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Выводы

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

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

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

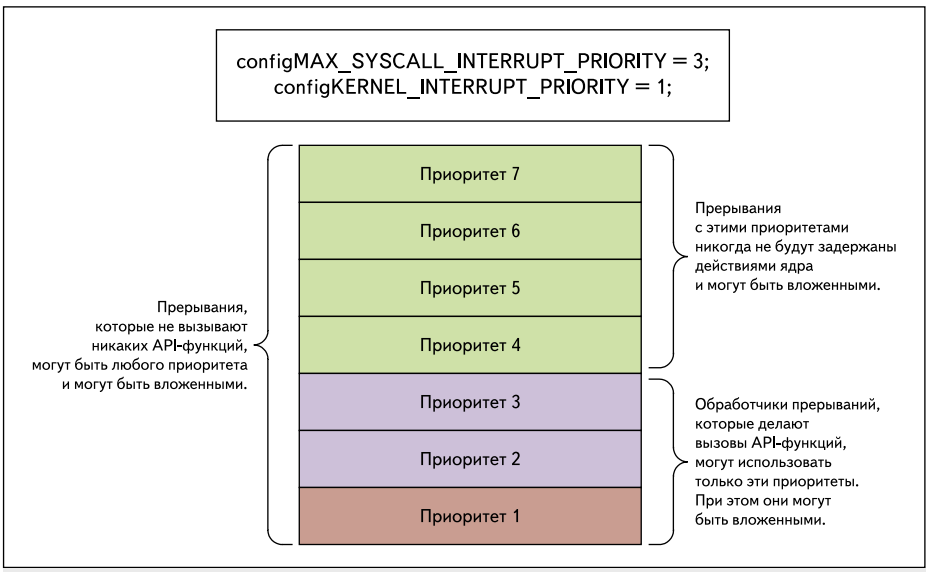

Рис. 13. Возможность вызова API-функций в обработчиках прерываний

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

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

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

Литература

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

Продолжение. Начало в № 2`2011

FreeRTOS — операционная система для микроконтроллеров

Андрей КУРНИЦ
kurnits@stim.by

Эта статья продолжает знакомить читателя с созданием программ, работающих под управлением FreeRTOS — операционной системы для микроконтроллеров. На этот раз речь пойдет о проблемах организации совместного доступа нескольких задач и/или прерываний к одному ресурсу в среде FreeRTOS.

Введение

Статья поможет читателям ответить на следующие вопросы:
1) Что означает термин «ресурс»?
2) Когда и почему необходимо управление доступом к ресурсам?
3) Что такое механизм взаимного исключения и способы его реализации?
4) Что такое критическая секция и способы ее реализации во FreeRTOS?
5) Как применять мьютексы для реализации механизма взаимного исключения?
6) Что такое инверсия приоритетов и как наследование приоритетов позволяет уменьшить (но не устранить) ее воздействие?
7) Другие потенциальные проблемы, возникающие при использовании мьютексов.
8) Задачи-сторожа — создание и использование.
9) Функция, вызываемая каждый системный квант времени.

Ресурсы и доступ к ним

Под ресурсами микроконтроллерной системы понимают как физически существующие устройства внутри микроконтроллера (области оперативной памяти и периферийные устройства), так и внешние по отношению к микроконтроллеру устройства (другие микроконтроллеры, контроллеры протоколов, дисплеи и т. д.). К этим группам можно свести все примеры ресурсов, приводимые ниже.

Потенциальная причина сбоев и ошибок в мультизадачных системах — это неправильно организованный совместный доступ к ресурсам из нескольких задач и/или прерываний. Одна задача получает доступ к ресурсу, начинает выполнять некоторые действия с ним, но не завершает операции с ресурсом до конца. В этот момент может произойти:

- Переключение контекста задачи, то есть процессор начнет выполнять другую задачу.
- Прерывание действия микроконтроллера, вследствие чего процессор займется выполнением обработчика соответствующего прерывания.

Если другая задача или обработчик возникшего прерывания обратятся к этому же самому ресурсу, состояние которого носит промежуточный характер из-за воздействия первой задачи, то результат работы программы будет отличаться от ожидаемого. Рассмотрим несколько примеров.

Доступ к внешней периферии

Рассмотрим сценарий, когда две задачи — задача А и задача Б — выводят информацию на ЖКИ-дисплей. Задача А ответственна за вывод значения каких-либо параметров на дисплей. Задача Б отвечает за вывод экстренных сообщений об авариях:

- Выполняется задача А и начинает выводить очередной параметр на дисплей: «Температура = 25 °С».
- Задача А вытесняется задачей Б в момент, когда на дисплей выведено лишь «Темпе».
- Задача Б выводит на дисплей экстренное сообщение «Превышено давление!!!», после чего переходит в блокированное состояние.
- Задача А возобновляет свое выполнение и выводит оставшуюся часть сообщения на дисплей: «ратура = 25 °С».

В итоге на дисплее появится искаженное сообщение: «ТемпеПревышено давление!!! ратура = 25 °С».

Неатомарные операции чтение/модификация/запись

Пусть стоит задача установить (сбросить, инвертировать — не имеет значения) один бит в регистре специальных функций, в данном случае — в регистре порта ввода/вывода микроконтроллера. Рассмотрим пример кода на языке Си и полученную в результате компиляции последовательность инструкций ассемблера.

Для микроконтроллеров AVR:

```
/* Код на Си */
PORTG ^= (1 << PG3);
/* Скомпилированный машинный код и инструкции ассемблера */
544:  80 91 65 00    lds   r24, 0x0065    ; Загрузить PORTG в регистр общего назначения
548:  98 e0          ldi   r25, 0x08      ; Бит PG3 — в другой регистр
54a:  89 27          eor   r24, r25       ; Операция Исключающее ИЛИ
54c:  80 93 65 00    sts   0x0065, r24    ; Результат — обратно в PORTG
```

Для микроконтроллеров ARM7:

```
/* Код на Си. */
PORTA |= 0x01;
/* Скомпилированный машинный код и инструкции ассемблера */
0x00000264 481C LDR R0,[PC,#0x0070]    ; Получить адрес PORTA
0x00000266 6801 LDR R1,[R0,#0x00]      ; Считать значение PORTA в R1
0x00000268 2201 MOV R2,#0x01           ; Поместить 1 в R2
0x0000026A 4311 ORR R1,R               ; Лог. И регистра R1 (PORTA) и R2 (константа 1)
0x0000026C 6001 STR R1,[R0,#0x00]      ; Сохранить новое значение в PORTA
```

И в первом, и во втором случае последовательность действий сводится:

- к копированию значения порта микроконтроллера в регистр общего назначения,
- к модификации регистра общего назначения,
- к обратному копированию результата из регистра общего назначения в порт.

Такую последовательность действий называют операцией чтения/модификации/записи.

Теперь рассмотрим случай, когда сразу две задачи выполняют операцию чтения/модификации/записи одного и того же порта.
1) Задача А загружает значение порта в регистр.

2) В этот момент ее вытесняет задача Б, при этом задача А не «успела» модифицировать и записать данные обратно в порт.
3) Задача Б изменяет значение порта и, например, блокируется.
4) Задача А продолжает выполняться с точки, в которой ее выполнение было прервано. При этом она продолжает работать с копией порта в регистре, выполняет какие-то действия над ним и записывает значение регистра обратно в порт.

Можно видеть, что в этом случае результат воздействия задачи Б на порт окажется потерян и порт будет содержать неверное значение.

О подобных операциях чтение/модификация/запись говорят, что они не являются атомарными. Атомарными же операциями называют те, выполнение которых не может быть прервано планировщиком. Приводя пример из архитектуры AVR, можно назвать инструкции процессора cbi и sbi, позволяющие сбросить/установить бит в регистре специальных функций. Разумеется, операция длиной в одну машинную инструкцию не может быть прервана планировщиком, то есть является атомарной.

Неатомарными могут быть не только операции с регистрами специальных функций. Операция над любой переменной языка Си, физический размер которой превышает разрядность микроконтроллера, является неатомарной. Например, операция инкремента глобальной переменной типа unsigned long на 8-битной архитектуре AVR выглядит так:

```
/* Код на Си */
unsigned long counter = 0;
counter++;
/* Скомпилированный машинный код и инструкции ассемблера */
618:   80 91 13 01      lds     r24, 0x0113
61c:   90 91 14 01      lds     r25, 0x0114
620:   a0 91 15 01      lds     r26, 0x0115
624:   b0 91 16 01      lds     r27, 0x0116
628:   01 96      adiw  r24, 0x01   ; 1
62a:   a1 1d      adc   r26, r1
62c:   b1 1d      adc   r27, r1
62e:   80 93 13 01      sts     0x0113, r24
632:   90 93 14 01      sts     0x0114, r25
636:   a0 93 15 01      sts     0x0115, r26
63a:   b0 93 16 01      sts     0x0116, r27
```

Если другая задача или прерывание обратятся к этой же переменной в течение этих 11 инструкций, результат окажется искаженным.

Следует отметить, что неатомарными являются также операции с составными типами — структурами, когда модифицируется сразу несколько членов структуры.

Реентерабельность функций

Функция называется реентерабельной, если она корректно работает при одновременном ее вызове из нескольких задач и/или прерываний. Под одновременным вызовом понимается вызов функции из одной задачи в тот момент, когда та уже вызвана из другой задачи, но еще не выполнена до конца.

Во FreeRTOS каждая задача имеет свой собственный стек и свой набор значений регистров процессора. Если функция использует переменные, расположенные только в стеке или в регистрах процессора, то она является реентерабельной. Напротив, функция, которая сохраняет свое состояние между вызовами в статической или глобальной переменной, не является реентерабельной.

Таким образом, функция, которая зависит только от своих параметров, не использует глобальные и статические переменные и вызывает только реентерабельные функции, будет реентерабельной [4].

Одновременный вызов нереентерабельной функции из нескольких задач может привести к непредсказуемому результату. Реентерабельными функциями можно пользоваться, не опасаясь одновременного их вызова из нескольких задач.

Рассмотрим пример реентерабельной функции:

```
/* Параметр передается в функцию через регистр общего назначения или стек. Это безопасно, т. к. каждая задача имеет свой набор регистров и свой стек. */
long lAddOneHundered( long lVar1 )
{
/* Объявлена локальная переменная. Компилятор расположит ее или в регистре или в стеке в зависимости от уровня оптимизации. Каждая задача и каждое прерывание, вызывающее эту функцию, будет иметь свою копию этой локальной переменной. */
long lVar2;
/* Какие-то действия над аргументом и локальной переменной. */
lVar2 = lVar1 + 100;
/* Обычно возвращаемое значение также помещается либо в стек, либо в регистр. */
return lVar2;
}
```

Теперь рассмотрим несколько нереентерабельных функций:

```
/* В этом случае объявлена глобальная переменная. Каждая задача, вызывающая функцию, которая использует эту переменную, будет «иметь дело» с одной и той же копией этой переменной */
long lVar1;

/* Нереентерабельная функция 1 */
long lNonReentrantFunction1( void )
{
/* Какие-то действия с глобальной переменной. */
lVar1 += 10;

return lVar1;
}

/* Нереентерабельная функция 2 */
void lNonReentrantFunction2( void )
{
/* Переменная, объявленная как статическая. Компилятор расположит ее не в стеке. Значит, каждая задача, вызывающая эту функцию, будет «иметь дело» с одной и той же копией этой переменной. */
static long lState = 0;
        switch( lState ) { /* … */};
}

/* Нереентерабельная функция 3 */
long lNonReentrantFunction3( void )
{
/* Функция, которая вызывает нереентерабельную функцию, также является нереентерабельной. */
return lNonReentrantFunction1() + 100;
}
```

Механизм взаимного исключения

Доступ к ресурсу, операции с которым одновременно выполняют несколько задач и/или прерываний, должен контролироваться механизмом взаимного исключения (mutual exclusion).

Механизм взаимного исключения гарантирует, что если задача начала выполнять некоторые действия с ресурсом, то никакая другая задача (или прерывание) не сможет получить доступ к данному ресурсу, пока операции с ним не будут завершены первой задачей.

FreeRTOS предлагает несколько способов реализации механизма взаимного исключения:

- критические секции;
- мьютексы;
- задачи-сторожа.

Однако наилучшая реализация взаимного исключения — это написание программы, в которой ресурсы не разделяются между несколькими задачами и доступ к одному ресурсу выполняет единственная задача или прерывание.

Критические секции

Сразу следует отметить, что критические секции — это очень грубый способ реализации взаимного исключения.

Критическая секция — это часть программы, которую в один момент времени может выполнять только одна задача или прерывание. Обычно защищаемый критической секцией участок кода начинается с инструкции входа в критическую секцию и заканчивается инструкцией выхода из нее.

Во FreeRTOS, в отличие от более сложных операционных систем, существует одна глобальная критическая секция. Если одна задача вошла в критическую секцию, то никакая другая задача не будет выполняться, пока не произойдет выход из критической секции.

FreeRTOS допускает два способа реализации критической секции:

- запрет прерываний;
- приостановка планировщика.

Запрет прерываний

Во FreeRTOS вход в критическую секцию, реализованную запретом прерываний, сводится к запрету всех прерываний процессора или (в зависимости от конкретного порта FreeRTOS) к запрету прерываний с приоритетом равным и ниже макроопределения ***configMAX_SYSCALL_INTERRUPT_PRIORITY***.

Во FreeRTOS участок кода, защищаемый критической секцией, которая реализована запретом прерываний, — это участок кода, окруженный вызовом API-макросов: ***taskENTER_CRITICAL()*** — вход в критическую секцию и ***taskEXIT_CRITICAL()*** — выход из критической секции.

Переключение контекста при вытесняющей многозадачности происходит по прерыванию (обычно от таймера), поэтому задача, которая вызвала ***taskENTER_CRITICAL()***, будет оставаться в состоянии выполнения, пока не вызовет ***taskEXIT_CRITICAL()***.

Участки кода, находящиеся внутри критической секции, должны быть как можно короче и выполняться как можно быстрее. Иначе использование критических секций негативно скажется на времени реакции системы на прерывания.

FreeRTOS допускает вложенный вызов макросов ***taskENTER_CRITICAL()*** и ***taskEXIT_CRITICAL()***, их реализация позволяет сохранять глубину вложенности. Выход программы из критической секции происходит, только если глубина вложенности станет равной нулю. Каждому вызову ***taskENTER_CRITICAL()*** должен соответствовать вызов ***taskEXIT_CRITICAL()***.

Пример использования критической секции:

```
/* Чтобы доступ к порту PORTA не был прерван никакой другой задачей, входим в критическую секцию. */
taskENTER_CRITICAL();
/* Переключение на другую задачу не может произойти, когда выполняется код, окруженный вызовом taskENTER_CRITICAL() и taskEXIT_CRITICAL().
Прерывания здесь могут происходить, только если микроконтроллер допускает вложение прерываний. Прерывание выполнится, если его приоритет выше константы configMAX_SYSCALL_INTERRUPT_PRIORITY. Однако такие прерывания не могут вызывать FreeRTOS API-функции. */
PORTA |= 0x01;
/* Неатомарная операция чтение/модификация/запись завершена. Сразу после этого выходим из критической секции. */
taskEXIT_CRITICAL();
```

Рассматривая пример выше, следует отметить, что если внутри критической секции произойдет прерывание с приоритетом выше ***configMAX_SYSCALL_INTERRUPT_PRIORITY***, которое, в свою очередь, обратится к порту PORTA, то принцип взаимного исключения доступа к ресурсу будет нарушен.

Приостановка/запуск планировщика

Еще один способ реализации критической секции в FreeRTOS — это приостановка работы планировщика (suspending the scheduler).

В отличие от реализации критической секции с помощью запрета прерываний (макросы ***taskENTER_CRITICAL()*** и ***taskEXIT_CRITICAL()***), которые защищают участок кода от доступа как из задач, так и из прерываний, реализация с помощью приостановки планировщика защищает участок кода только от доступа из другой задачи. Все прерывания микроконтроллера остаются разрешены.

Операция запуска планировщика после приостановки выполняется существенно дольше макроса ***taskEXIT_CRITICAL()***, это немаловажно с точки зрения сокращения времени выполнения критических секций в программе. Этот момент следует учитывать при выборе способа организации критических секций.

Приостановка планировщика выполняется API-функцией ***vTaskSuspendAll()***. Ее прототип:

```
void vTaskSuspendAll( void );
```

После вызова ***vTaskSuspendAll()*** планировщик останавливается, переключения контекста каждый системный квант времени не происходит, задача, которая вызвала ***vTaskSuspendAll()***, будет выполняться сколь угодно долго до запуска планировщика. API-функция ***vTaskSuspendAll()*** не влияет на прерывания: если до вызова ***vTaskSuspendAll()*** они были разрешены, то при возникновении прерываний их обработчики будут выполняться.

Если же обработчик прерывания выполнил макрос принудительного переключения контекста (***portSWITCH_CONTEXT()***, ***taskYIELD()***, ***portYIELD_FROM_ISR()*** и др.— в зависимости от порта FreeRTOS), то запрос на переключение контекста будет выполнен, как только работа планировщика будет возобновлена.

Другие API-функции FreeRTOS нельзя вызывать, когда планировщик приостановлен вызовом ***vTaskSuspendAll()***.

Для возобновления работы планировщика служит API-функция ***xTaskResumeAll()***, ее прототип:

```
portBASE_TYPE xTaskResumeAll( void );
```

Возвращаемое значение может быть равно:

- ***pdTRUE*** — означает, что переключение контекста произошло сразу после возобновления работы планировщика.
- ***pdFALSE*** — во всех остальных случаях.

Возможен вложенный вызов ***API-функций vTaskSuspendAll()*** и ***xTaskResumeAll()***. При этом ядро автоматически подсчитывает глубину вложенности. Работа планировщика будет возобновлена, если глубина вложенности станет равна 0. Этого можно достичь, если каждому вызову ***vTaskSuspendAll()*** будет соответствовать вызов ***xTaskResumeAll()***.

Мьютексы

Взаимное исключение называют также мьютексом (mutex — MUTual EXclusion), этот термин чаще используется в операционных системах Windows и Unix-подобных [5].

Мьютекс во FreeRTOS представляет собой специальный тип двоичного семафора, который используется для реализации совместного доступа к ресурсу двух или большего числа задач. При использовании в качестве механизма взаимного исключения мьютекс можно представить как семафор, относящийся к ресурсу, доступом к которому необходимо управлять.

В отличие от семафора мьютекс во FreeRTOS предоставляет механизм наследования приоритетов, о котором будет рассказано ниже. Также следует отметить, что использование мьютекса из тела обработчика прерывания невозможно.

Чтобы корректно получить доступ к ресурсу, задача должна предварительно захватить мьютекс, стать его владельцем. Когда владелец семафора закончил операции с ресурсом, он должен отдать мьютекс обратно. Только когда мьютекс освободился (возвращен какой-либо задачей), другая задача может его захватить и безопасно выполнить свои операции с общим для нескольких задач ресурсом. Задаче не разрешено выполнять операции с ресурсом, если в данный момент она не является владельцем мьютекса. Процессы, происходящие при взаимном исключении доступа с использованием мьютекса, приведены на рис. 1.

Обе задачи нуждаются в доступе к ресурсу, однако только задача-владелец мьютекса может его получить (рис. 1а). Задача А пытается захватить мьютекс, в этот момент он свободен, поэтому она становится его владельцем (рис. 1б). Задача А выполняет некоторые действия с ресурсом. В этот момент задача Б пытается захватить тот же самый мьютекс, однако это ей не удается, потому что задача А все еще является его владельцем. Соответственно, пока задача А выполняет операции с ресурсом, задача Б не может получить к нему доступ и переходит в блокированное состояние (рис. 1в). Задача А до конца завершает операции с ресурсом и возвращает мьютекс обратно (рис. 1г). Это приводит к разблокировке задачи Б, теперь она получает доступ к ресурсу (рис. 1д). При завершении действий с ресурсом задача Б обязана отдать мьютекс обратно (рис. 1е).

Легко заметить, что мьютексы и двоичные семафоры очень похожи в использовании. Отличие заключается в том, что мьютекс после захвата обязательно должен быть возвращен, иначе другие задачи не смогут получить доступ к разделяемому ресурсу. Двоичный семафор, используемый в целях синхронизации выполнения задач (и прерываний), наоборот — не должен возвращаться задачей, которая его захватила.

Важным моментом является то, что непосредственно мьютекс не защищает ресурс от одновременного доступа нескольких задач. Вместо этого реализация всех задач в системе должна быть выполнена так, чтобы перед инструкцией доступа к ресурсу следовал вызов API-функции захвата соответствующего мьютекса. Эта обязанность ложится на программиста.

Работа с мьютексами

Мьютекс представляет собой специальный вид семафора, поэтому доступ к мьютексу осуществляется так же, как и к семафору: с помощью дескриптора (идентификатора) мьютекса — переменной типа ***xSemaphoreHandle***.

Для того чтобы API-функции для работы с мьютексами были включены в программу, необходимо установить макроопределение ***configUSE_MUTEXES*** в файле ***FreeRTOSConfig.h*** равным «1».

Мьютекс должен быть явно создан перед первым его использованием. API-функция ***xSemaphoreCreateMutex()*** служит для создания мьютекса:

```
xSemaphoreHandle xSemaphoreCreateMutex( void );
```

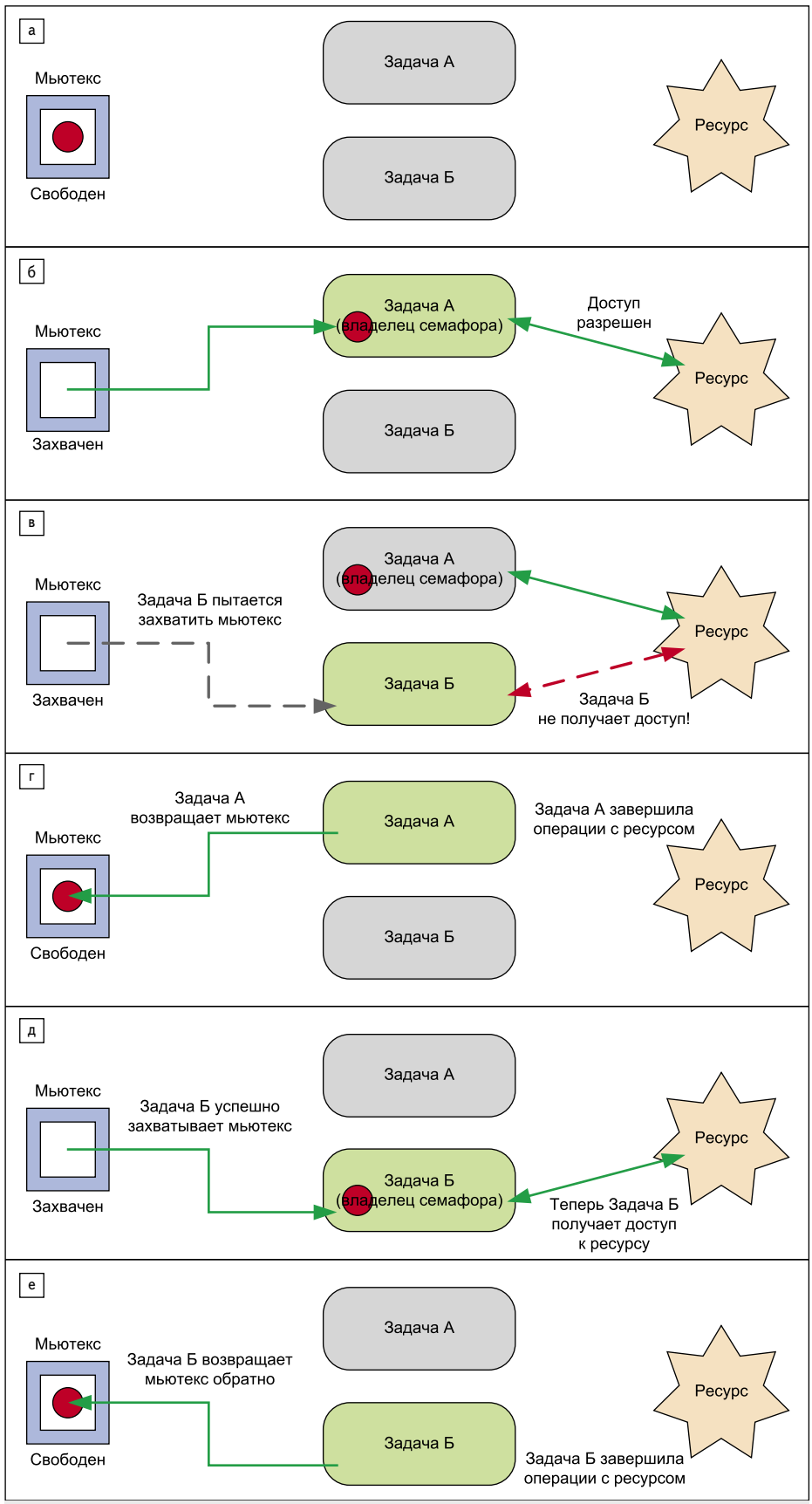

Рис. 1. Использование мьютекса для управления доступом к ресурсу

Возвращаемое значение — дескриптор мьютекса, он должен быть сохранен в переменной для дальнейшего обращения к мьютексу. Если мьютекс не создан по причине отсутствия достаточного объема памяти, возвращаемым значением будет NULL.

Операции захвата и возврата (выдачи) мьютекса выполняются с помощью аналогичных API-функций для работы с семафорами — ***xSemaphoreTake()*** и ***xSemaphoreGive()***, которые были рассмотрены в [1, № 7].

Рассмотрим, как применение мьютекса позволяет решить проблему совместного доступа к ресурсу, на примере учебной программы № 1. В качестве разделяемого ресурса выступает консоль, две задачи выводят свое сообщение на дисплей. Обратите внимание на реализацию вывода строки на консоль: вместо стандартной функции используется посимвольный вывод.

Сначала рассмотрим учебную программу № 1 без использования мьютекса:

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

/* Дескриптор мьютекса — глобальная переменная*/
volatile xSemaphoreHandle xMutex;

/* Функция посимвольно выводит строку на консоль.
Консоль, как ресурс, никаким образом не защищена от совместного
доступа из нескольких задач. */
static void prvNewPrintString(const portCHAR *pcString) {
   portCHAR *p;
   int i;

   /* Указатель — на начало строки */
   p = pcString;
   /* Пока не дошли до нулевого символа — конца строки. */
   while (*p) {
      /* Вывод на консоль символа, на который ссылается указатель. */
      putchar(*p);
      /* Указатель — на следующий символ в строке. */
      p++;
      /* Вывести содержимое буфера экрана на экран. */
      fflush(stdout);
      /* Небольшая пауза */
      for (i = 0; i < 10000; i++) ;
   }
}

/* Функция, реализующая задачу.
 Будет создано 2 экземпляра этой задачи.
 Каждый получит строку символов в качестве аргумента
 при создании задачи. */
static void prvPrintTask(void *pvParameters) {
      char *pcStringToPrint;
      pcStringToPrint = (char *) pvParameters;
      for (;;) {
         /* Для вывода строки на консоль используется своя
            функция prvNewPrintString(). */
         prvNewPrintString(pcStringToPrint);
         /* Блокировать задачу на промежуток времени случайной
            длины: от 0 до 500 мс. */
         vTaskDelay((rand() % 500));
         /* Вообще функция rand() не является реентерабельной.
            Однако в этой программе это неважно. */
      }
}

/*-----------------------------------------------------------*/
/* Точка входа. С функции main() начнется выполнение
   программы. */
short main( void )
{
      /* Создание мьютекса. */
      xMutex = xSemaphoreCreateMutex();
      /* Создание задач, если мьютекс был успешно создан. */
      if (xMutex != NULL) {
         /* Создать два экземпляра одной задачи. Каждому
            экземпляру задачи передать в качестве аргумента свою
            строку. Приоритет задач задать разным, чтобы имело
            место вытеснение задачи 1 задачей 2.
      */ xTaskCreate(prvPrintTask, "Print1", 1000,
               "Task 1 ******************************************\r\n", 1,
            NULL);
         xTaskCreate(prvPrintTask, "Print2", 1000,
               "Task 2 ---------------------------------\r\n", 2,
            NULL);
            /* Запуск планировщика. */
            vTaskStartScheduler();
      }
      return 1;
}
```

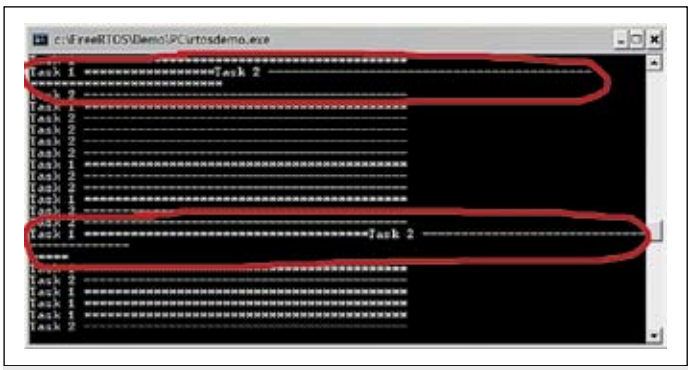

Рис. 2. Результат работы учебной программы № 1 без использования мьютекса

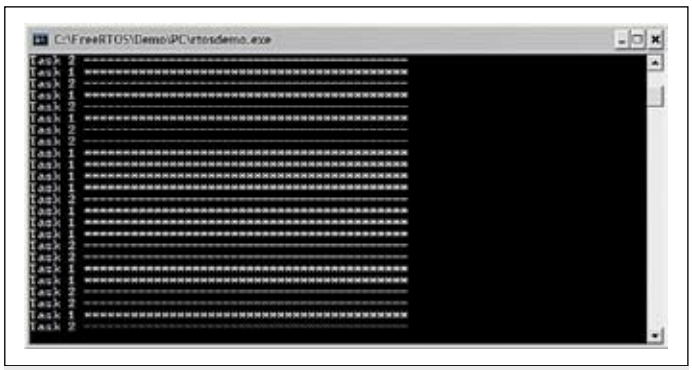

Рис. 3. Результат работы учебной программы № 1 с применением мьютекса

Как видно по результатам работы (рис. 2), совместный доступ к консоли без применения какого-либо механизма взаимного исключения приводит к тому, что некоторые сообщения, которые выводят на консоль задачи, оказываются повреждены.

Теперь защитим консоль от одновременного доступа с помощью мьютекса, заменив реализацию функции ***prvNewPrintString()*** на следующую:

```
/* Функция посимвольно выводит строку на консоль.
Консоль, как ресурс, защищена от совместного доступа
из нескольких задач с помощью мьютекса. */
static void prvNewPrintString(const portCHAR *pcString) {
   portCHAR *p;
   int i;

   /* Указатель — на начало строки */
   p = pcString;
   /* Захватить мьютекс. Время ожидания в блокированном
    состоянии, если мьютекс недоступен, сколь угодно долго.
    Возвращаемое значение xSemaphoreTake() должно проверяться,
    если указано время пребывания в блокированном состоянии,
    отличное от portMAX_DELAY */
   xSemaphoreTake( xMutex, portMAX_DELAY ); {
      /* Пока не дошли до нулевого символа — конца строки. */
      while (*p) {
         /* Вывод на консоль символа, на который ссылается указатель. */
         putchar(*p);
         /* Указатель — на следующий символ в строке. */
         p++;
         /* Вывести содержимое буфера экрана на экран. */
         fflush(stdout);
         /* Небольшая пауза */
         for (i = 0; i < 10000; i++) ;
      }
   }
   /* Когда вывод ВСЕЙ строки на консоль закончен,
    освободить мьютекс. Иначе другие задачи не смогут
    обратиться к консоли! */
   xSemaphoreGive( xMutex );
}
```

Теперь при выполнении учебной программы № 1 сообщения от разных задач не накладываются друг на друга: совместный доступ к ресурсу (консоли) организован правильно (рис. 3).

Рекурсивные мьютексы

Помимо обычных мьютексов, рассмотренных выше, FreeRTOS поддерживает также рекурсивные мьютексы (Recursive Mutexes) [6]. Их основное отличие от обычных мьютексов заключается в том, что они корректно работают при вложенных операциях захвата и освобождения мьютекса. Вложенные операции захвата/освобождения мьютекса допускаются только в теле задачи-владельца мьютекса. Рассмотрим пример рекурсивного захвата обычного мьютекса:

```
xSemaphoreHandle xMutex;
/* ... */
xMutex = xSemaphoreCreateMutex();
/* ... */

/* Функция, реализующая задачу. */
void vTask(void *pvParameters) {
   for (;;) {
      /* Захват мьютекса */
      xSemaphoreTake( xMutex, portMAX_DELAY );
      /* Действия с ресурсом */
      /* ... */
      /* Вызов функции, которая выполняет операции с этим же
         ресурсом. */
      vSomeFunction();
      /* Действия с ресурсом */
      /* ... */
      /* Действия с ресурсом закончены. Освободить мьютекс. */
      xSemaphoreGive( xMutex );
   }
}

/* Функция, которая вызывается из тела задачи vTask*/
void vSomeFunction(void) {
   /* Захватить тот же самый мьютекс.
      Т. к. тайм-аут не указан, то задача «зависнет» в ожидании,
      пока мьютекс не освободится.
      Однако это никогда не произойдет! */
   if (xSemaphoreTake( xMutex, portMAX_DELAY ) == pdTRUE )
{
      /* Действия с ресурсом внутри функции */
      /* ... */
      /* Освободить мьютекс */
      xSemaphoreGive( xMutex );
   }
}
```

Такое использование обычного мьютекса приведет к краху программы. При попытке повторно захватить мьютекс внутри функции ***vSomeFunction()*** задача ***vTask*** перейдет в блокированное состояние, пока мьютекс не будет возвращен. Другие задачи смогут выполнить возврат мьютекса только после того, как сами его захватят. Однако мьютекс уже захвачен, поэтому задача ***vTask*** заблокируется на бесконечно долгое время («зависнет»).

Если при повторном вызове API-функции ***xSemaphoreTake()*** было указано конечное время тайм-аута, «зависания» задачи не произойдет. Вместо этого действия с ресурсом, выполняемые внутри функции, никогда не будут произведены, что также является недопустимой ситуацией.

Когда программа проектируется так, что операции захват/освобождение мьютекса являются вложенными, следует использовать рекурсивные мьютексы:

```
xSemaphoreHandle xRecursiveMutex;
/* ... */
xRecursiveMutex = xSemaphoreCreateRecursiveMutex();
/* ... */

/* Функция, реализующая задачу. */
void vTask(void *pvParameters) {
   for (;;) {
      /* Захват мьютекса */
      xSemaphoreTakeRecursive( xRecursiveMutex, portMAX_DELAY );
      /* Действия с ресурсом */
      /* ... */
      /* Вызов функции, которая выполняет операции с этим же
         ресурсом. */
      vSomeFunction();
      /* Действия с ресурсом */
      /* ... */
      /* Действия с ресурсом закончены. Освободить мьютекс. */
      xSemaphoreGiveRecursive( xRecursiveMutex );
   }
}

/* Функция, которая вызывается из тела задачи vTask*/
void vSomeFunction(void) {
   /* Захватить тот же самый мьютекс.
      При этом состояние мьютекса никоим образом не изменится.
      Задача не заблокируется, действия с ресурсом внутри этой
      функции будут выполнены. */
   if (xSemaphoreTakeRecursive( xRecursiveMutex, portMAX_DELAY
) == pdTRUE ) {
      /* Действия с ресурсом внутри функции */
      /* ... */
      /* Освободить мьютекс */
      xSemaphoreGiveRecursive( xRecursiveMutex );
   }
}
```

В этом случае программа будет работать корректно. При повторном захвате мьютекса API-функцией ***xSemaphoreTakeRecursive()*** задача не перейдет в блокированное состояние, и эта же задача останется владельцем мьютекса. Вместо этого увеличится на единицу внутренний счетчик, который определяет, сколько операций «захват» было применено к мьютексу, действия с ресурсом внутри функции ***vSomeFunction()*** будут выполнены, так как задача ***vTask*** остается владельцем мьютекса. При освобождении мьютекса (при вызове API-функции ***xSemaphoreGiveRecursive()***) из тела одной и той же задачи внутренний счетчик уменьшается на единицу. Когда этот счетчик станет равен нулю, это будет означать, что текущая задача больше не является владельцем мьютекса и теперь он может быть захвачен другой задачей.

Таким образом, каждому вызову API-функции ***xSemaphoreTakeRecursive()*** внутри тела одной и той же задачи должен соответствовать вызов API-функции ***xSemaphoreGiveRecursive()***.

Для того чтобы использовать рекурсивные мьютексы в программе, необходимо установить макроопределение ***configUSE_RECURSIVE_MUTEXES*** в файле ***FreeRTOSConfig.h*** равным «1».

Как и обращение к обычному мьютексу, обращение к рекурсивному мьютексу осуществляется с помощью дескриптора (идентификатора) мьютекса — переменной типа ***xSemaphoreHandle***.

API-функции для работы с рекурсивными мьютексами:

- ***xSemaphoreCreateRecursiveMutex()*** — создание рекурсивного мьютекса;
- ***xSemaphoreTakeRecursive()*** — захват рекурсивного мьютекса;
- ***xSemaphoreGiveRecursive()*** — освобождение (возврат) рекурсивного мьютекса.

Набор параметров и возвращаемое значение этих API-функций ничем не отличаются от соответствующих API-функций для работы с обычными мьютексами. Стоит помнить лишь о том, что API-функции для работы с рекурсивными мьютексами нельзя применять к обычным мьютексам и наоборот.

Проблемы при использовании мьютексов

Инверсия приоритетов

Вернемся к рассмотрению учебной программы № 1. Возможная последовательность выполнения задач приведена на рис. 4. Такая последовательность имела бы место, если во FreeRTOS не был бы реализован механизм наследования приоритетов, о котором будет рассказано ниже.

Пусть в момент времени (1) низкоприоритетная задача 1 вытеснила задачу Бездействие, так как закончился период пребывания задачи 1 в блокированном состоянии (рис. 4). Задача 1 захватывает мьютекс (становится его владельцем) и начинает посимвольно выводить свою строку на дисплей (2). В момент времени (3) разблокируется высокоприоритетная задача 2, при этом она вытесняет задачу 1, когда та еще не закончила вывод строки на дисплей. Задача 2 пытается захватить мьютекс, однако он уже захвачен задачей 1, поэтому задача 2 блокируется в ожидании, когда мьютекс станет доступен. Управление снова получает задача 1, она завершает вывод строки на дисплей — операция с ресурсом завершена. Задача 1 возвращает мьютекс обратно — мьютекс становится доступен (момент времени (4)). Как только мьютекс становится доступен, разблокируется задача 2, которая ожидала его освобождения. Задача 2 захватывает мьютекс (становится его владельцем) и выводит свою строку на дисплей. Приоритет задачи 2 выше, чем у задачи 1, поэтому задача 2 выполняется все время, пока полностью не выведет свою строку на дисплей, после чего она отдает мьютекс обратно и блокируется на заданное API-функцией vTaskDelay() время — момент времени (5). Задача 1 снова получает управление, но на непродолжительное время — пока также не перейдет в блокированное состояние, вызвав vTaskDelay().

Учебная программа № 1 и рис. 4 демонстрируют одну из возможных проблем, возникающих при использовании мьютексов для реализации механизма взаимного исключения, — проблему инверсии приоритетов (Priority Inversion) [7]. На рис. 4 представлена ситуация, когда высокоприоритетная задача 2 вынуждена ожидать, пока низкоприоритетная задача 1 завершит действия с ресурсом и возвратит мьютекс обратно. То есть на некоторое время фактический приоритет задачи 2 оказывается ниже приоритета задачи 1: происходит инверсия приоритетов.

В реальных программах инверсия приоритетов может оказывать еще более негативное влияние на выполнение высокоприоритетных задач. Рассмотрим пример. В программе могут существовать также задачи со «средним» приоритетом — ниже, чем у высокоприоритетной, которая ожидает освобождения мьютекса, но выше, чем у низкоприоритетной, которая в данный момент захватила мьютекс и выполняет действия с разделяемым ресурсом. Среднеприоритетные задачи могут разблокироваться на протяжении интервала, когда низкоприоритетная задача владеет мьютексом. Такой сценарий является

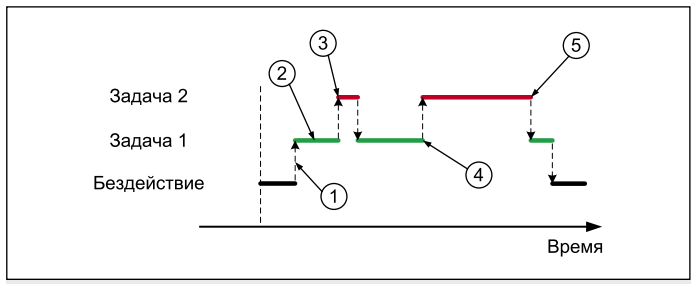

Рис. 4. Переключение между задачами в учебной программе № 1 без механизма наследования приоритетов

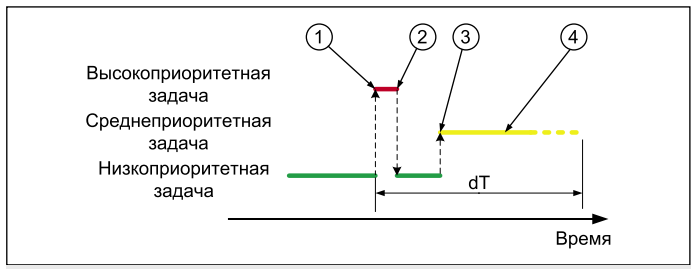

Рис. 5. Наихудший случай влияния инверсии приоритетов

наихудшим, так как ко времени, когда высокоприоритетная задача ожидает освобождения мьютекса, будет добавлено время выполнения среднеприоритетных задач (рис. 5).

Низкоприоритетная задача стала владельцем мьютекса ранее. Происходит некоторое событие, за обработку которого отвечает высокоприоритетная задача. Она разблокируется и пытается захватить мьютекс (1), это ей не удается, и она блокируется — момент времени (2) на рис. 5. Управление снова возвращается низкоприоритетной задаче, которая в момент времени (3) вытесняется задачей, приоритет которой выше (среднеприоритетной задачей). Среднеприоритетная задача может выполняться продолжительное время, в течение которого высокоприоритетная будет ожидать, пока мьютекс не будет освобожден низкоприоритетной задачей (4). Время реакции на событие при этом значительно удлиняется — величина dT на рис. 5.

В итоге инверсия приоритетов может значительно ухудшить время реакции микроконтроллерной системы на внешние события.

Для уменьшения (но не полного исключения) негативного влияния инверсии приоритетов во FreeRTOS реализован механизм наследования приоритетов (Priority Inheritance). Его работа заключается во временном увеличении приоритета низкоприоритетной задачи-владельца мьютекса до уровня приоритета высокоприоритетной задачи, которая в данный момент пытается захватить мьютекс. Когда низкоприоритетная задача освобождает мьютекс, ее приоритет уменьшается до значения, которое было до повышения. Говорят, что низкоприоритетная задача наследует приоритет высокоприоритетной задачи.

Рассмотрим работу механизма наследования приоритетов на примере программы с высоко-, средне- и низкоприоритетной задачами (рис. 6).

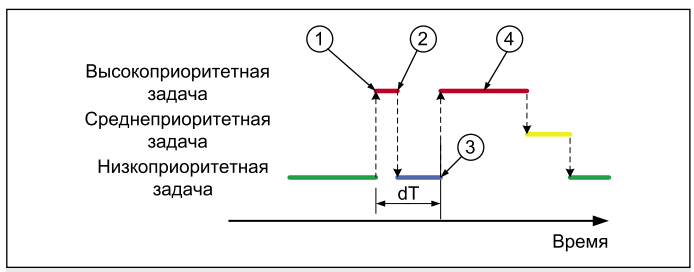

Рис. 6. Уменьшение влияния инверсии приоритетов при работе механизма наследования приоритетов

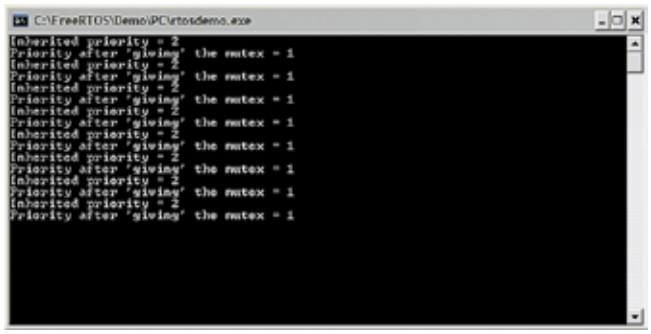

Рис. 7. Результат работы учебной программы № 2

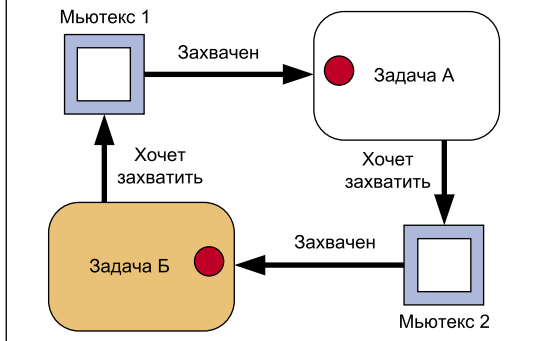

Рис. 8. Взаимная блокировка двух задач

Низкоприоритетная задача стала владельцем мьютекса ранее. Происходит некоторое событие, за обработку которого отвечает высокоприоритетная задача. Она разблокируется и пытается захватить мьютекс (1), это ей не удается, и она блокируется — момент времени (2) на рис. 6. Однако в результате попытки высокоприоритетной задачи захватить мьютекс низкоприоритетная задача-владелец мьютекса наследует приоритет этой высокоприоритетной задачи. Теперь низкоприоритетная задача не может быть вытеснена среднеприоритетной задачей. Поэтому в момент времени (3), когда низкоприоритетная задача завершила операции с ресурсом и возвращает мьютекс, разблокируется, захватывает мьютекс и начинает выполняться высокоприоритетная задача (4). Приоритет же низкоприоритетной задачи при этом возвращается к своему «нормальному» значению.

Таким образом, механизм наследования приоритетов уменьшает время реакции системы на событие, когда происходит инверсия приоритетов (сравните величину dT на рис. 5 и рис. 6).

Продемонстрировать работу механизма наследования приоритетов во FreeRTOS позволяет учебная программа № 2. В программе выполняются две задачи: низкоприоритетная задача 1 с приоритетом 1 и высокоприоритетная задача 2 с приоритетом 2. Обе задачи пытаются захватить один и тот же мьютекс. Низкоприоритетная задача сигнализирует на дисплей, если ее приоритет изменился (повысился):

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

/* Дескриптор мьютекса — глобальная переменная*/
volatile xSemaphoreHandle xMutex;

/* Низкоприоритетная задача 1. Приоритет = 1. */
static void prvTask1(void *pvParameters) {
  long i;
  /* Логическая переменная. Определяет, произошло ли
     наследование приоритетов. */
  unsigned portBASE_TYPE uxIsPriorityInherited = pdFALSE;
  /* Бесконечный цикл */
  for (;;) {
    /* Наследования приоритетов еще не было */
    uxIsPriorityInherited = pdFALSE;
    /* Захватить мьютекс. */
    xSemaphoreTake(xMutex, portMAX_DELAY);
    /* Какие-то действия. За это время высокоприоритетная
       задача попытается захватить мьютекс. */
    for (i = 0; i < 100000L; i++)
      ;
    /* Если приоритет этой задачи изменился (был унаследован
       от задачи 2). */
    if (uxTaskPriorityGet(NULL) != 1) {
      printf("Inherited priority = %d\n\r", uxTaskPriorityGet(NULL));
      uxIsPriorityInherited = pdTRUE;
    }
    /* Освободить мьютекс. */
    xSemaphoreGive(xMutex);
    /* Вывести значение приоритета ПОСЛЕ освобождения
       мьютекса. */
    if (uxIsPriorityInherited == pdTRUE) {
      printf("Priority after 'giving' the mutex = %d\n\r",
uxTaskPriorityGet(NULL));
    }
    /* Блокировать задачу на промежуток времени случайной
       длины: от 0 до 500 мс. */
    vTaskDelay((rand() % 500));
  }
}

/* Высокоприоритетная задача 2. Приоритет = 2. */
static void prvTask2(void *pvParameters) {
    for (;;) {
        xSemaphoreTake( xMutex, portMAX_DELAY );
        xSemaphoreGive( xMutex );
        /* Интервал блокировки короче — от 0 до 50 мс */
          vTaskDelay((rand() % 50));
    }
}

/*-----------------------------------------------------------*/
/* Точка входа. С функции main() начнется выполнение программы. */
short main( void )
{
    /* Создание мьютекса. */
    xMutex = xSemaphoreCreateMutex();
    /* Создание задач, если мьютекс был успешно создан. */
    if (xMutex != NULL) {
        xTaskCreate(prvTask1, "prvTask1", 1000, NULL, 1, NULL);
        xTaskCreate(prvTask2, "prvTask2", 1000, NULL, 2, NULL);
        /* Запуск планировщика. */
        vTaskStartScheduler();
    }
    return 1;
}
```

По результатам выполнения учебной программы № 2 (рис. 7) видно, что приоритет задачи 1 временно увеличивается со значения 1 до значения 2, когда задача 2 пытается захватить мьютекс, который уже захвачен задачей 1. После того как задача 1 освобождает мьютекс, ее приоритет возвращается к первоначальному значению.

Следует отметить, что механизм наследования приоритетов во FreeRTOS только уменьшает, однако не устраняет полностью негативное влияние инверсии приоритетов. Поэтому рекомендуется проектировать программу так, чтобы избегать ситуации инверсии приоритетов.

Взаимная блокировка

Взаимная блокировка (Deadlock или Deadly Embrace) — это ситуация в многозадачной системе, когда несколько задач находятся в состоянии бесконечного ожидания доступа к ресурсам, занятым самими этими задачами [8].

Простейший пример взаимной блокировки включает две задачи — задачу А и задачу Б и два мьютекса — мьютекс 1 и мьютекс 2. Взаимная блокировка может произойти при такой последовательности событий:

- Выполняется задача А, которая успешно захватывает мьютекс 1.
- Задача Б вытесняет задачу А.
- Задача Б успешно захватывает мьютекс 2, после чего пытается захватить и мьютекс 1. Это ей не удается, и она блокируется в ожидании освобождения мьютекса 1.
- Управление снова получает задача А. Она пытается захватить мьютекс 2, однако он уже захвачен задачей Б. Поэтому задача А блокируется в ожидании освобождения мьютекса 2.

В итоге получаем ситуацию, когда задача А заблокирована в ожидании освобождения мьютекса 2, захваченного задачей Б. Задача Б заблокирована в ожидании освобождения мьютекса 1, захваченного задачей А. Графически эта ситуация представлена на рис. 8.

Впрочем, в состояние взаимной блокировки может попасть любое количество задач, находящихся в круговой зависимости друг от друга. Если ситуация взаимной блокировки единожды наступила, то выход из этой ситуации невозможен.

Как и в случае с инверсией приоритетов, лучший способ избежать взаимной блокировки задач — это исключить такую возможность на этапе проектирования программы, то есть не создавать круговой зависимости задач друг от друга.

Следует отметить, что помимо рассмотренных выше проблем совместного доступа к ресурсам существуют еще такие, как голодание (Starvation) и разновидность взаимной блокировки, при которой задачи не блокируются, но и не выполняют полезной работы (Livelock). Подробнее с ними можно ознакомиться в [9].

Функция vApplicationTickHook()

Прежде чем продолжить изучение механизмов взаимного исключения, стоит обратить внимание на еще одну возможность FreeRTOS. Как известно, подсистема времени FreeRTOS [1, № 4] основывается на системном кванте времени. По прошествии каждого кванта времени ядро FreeRTOS выполняет внутренние системные действия, связанные как с работой планировщика, так и с отсчетом произвольных временных промежутков.

Программисту предоставляется возможность определить свою функцию, которая будет вызываться каждый системный квант времени. Такая возможность может оказаться полезной, например, для реализации механизма программных таймеров.

Чтобы задать свою функцию, которая будет вызываться каждый системный квант времени, необходимо в файле настроек ядра ***FreeRTOSConfig.h*** задать макроопределение ***configUSE_TICK_HOOK*** равным 1. Сама функция должна содержаться в программе и иметь следующий прототип:

```
void vApplicationTickHook( void );
```

Как и функция задачи Бездействие, функция ***vApplicationTickHook()*** является функцией-ловушкой или функцией обратного вызова (callback function). Поэтому в программе не должны встречаться явные вызовы этой функции.

Отсчет квантов времени во FreeRTOS реализован за счет использования прерывания от одного из аппаратных таймеров микроконтроллера, вследствие чего функция ***vApplicationTickHook()*** вызывается из обработчика прерывания. Поэтому к ней предъявляются следующие требования:

- Она должна выполняться как можно быстрее.
- Должна использовать как можно меньше стека.
- Не должна содержать вызовы API-функций, кроме предназначенных для вызова из обработчика прерывания (то есть чьи имена заканчиваются на ***FromISR*** или ***FROM_ISR***).

Задачи-сторожа (Gatekeeper tasks)

Задачи-сторожа предоставляют простой и прозрачный метод реализации механизма взаимного исключения, которому не присущи проблемы инверсии приоритетов и взаимной блокировки.

Задача-сторож — это задача, которая имеет единоличный доступ к разделяемому ресурсу. Никакая другая задача в программе не имеет права обращаться к ресурсу напрямую. Вместо этого все задачи, разделяющие общий ресурс, обращаются к задаче-сторожу, используя безопасные механизмы межзадачного взаимодействия FreeRTOS. Непосредственно действия с ресурсом выполняет задача-сторож.

В отличие от мьютексов, работать с которыми могут только задачи, к задаче-сторожу могут обращаться как задачи, так и обработчики прерываний.

Рассмотрим использование задачи-сторожа на примере учебной программы № 3. Как и в учебной программе № 1, здесь разделяемым ресурсом выступает консоль. В программе созданы две задачи, каждая из которых выводит свое сообщение на консоль. Кроме того, сообщения выводит функция, вызываемая каждый системный квант времени, это демонстрирует возможность обращения к разделяемому ресурсу из тела обработчика прерывания:

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

/* Прототип задачи, которая выводит сообщения на консоль,
   передавая их задаче-сторожу.
 * Будет создано 2 экземпляра этой задачи */
static void prvPrintTask(void *pvParameters);

/* Прототип задачи-сторожа */
static void prvStdioGatekeeperTask(void *pvParameters);

/* Таблица строк, которые будут выводиться на консоль */
static char *pcStringsToPrint[] = {
   "Task 1 ****************************************************\r\
n",
   "Task 2 ----------------------------------------------------\r\n",
   "Message printed from the tick hook interrupt ##############\r\n"
};

/*-----------------------------------------------------------*/

/* Объявить очередь, которая будет использоваться для передачи
   сообщений от задач и прерываний к задаче-сторожу. */
xQueueHandle xPrintQueue;

int main(void) {
   /* Создать очередь длиной макс. 5 элементов типа
      "указатель на строку" */
   xPrintQueue = xQueueCreate(5, sizeof(char *));

   /* Проверить, успешно ли создана очередь. */
   if (xPrintQueue != NULL) {
      /* Создать два экземпляра задачи, которые будут выводить
         строки на консоль, передавая их задаче-сторожу.
         В качестве параметра при создании задачи передается
         номер строки в таблице. Задачи создаются с разными
         приоритетами. */
      xTaskCreate(prvPrintTask, "Print1", 1000, (void *) 0, 1, NULL);
      xTaskCreate(prvPrintTask, "Print2", 1000, (void *) 1, 2, NULL);

      /* Создать задачу-сторож. Только она будет иметь
         непосредственный доступ к консоли. */
      xTaskCreate(prvStdioGatekeeperTask, "Gatekeeper", 1000, NULL,
0, NULL);

      /* Запуск планировщика. */
      vTaskStartScheduler();
   }
   return 0;
}
/*-----------------------------------------------------------*/

static void prvStdioGatekeeperTask(void *pvParameters) {
   char *pcMessageToPrint;

   /* Задача-сторож. Только она имеет прямой доступ к консоли.
    * Когда другие задачи "хотят" вывести строку на консоль,
      они записывают указатель на нее в очередь.
    * Указатель из очереди считывает задача-сторож
      и непосредственно выводит строку */
   for (;;) {
      /* Ждать появления сообщения в очереди. */
      xQueueReceive(xPrintQueue, &pcMessageToPrint, portMAX_
DELAY);
      /* Непосредственно вывести строку. */
      printf("%s", pcMessageToPrint);
      fflush(stdout);
      /* Вернуться к ожиданию следующей строки. */
   }
}
/*-----------------------------------------------------------*/

/* Задача, которая автоматически вызывается каждый системный
   квант времени.
 * Макроопределение configUSE_TICK_HOOK должно быть равно 1. */
void vApplicationTickHook(void) {
   static int iCount = 0;
   portBASE_TYPE xHigherPriorityTaskWoken = pdFALSE;

   /* Выводить строку каждые 200 квантов времени.
      Строка не выводится напрямую, указатель на нее помещается
      в очередь и считывается задачей-сторожем. */
   iCount++;
   if (iCount >= 200) {
      /* Используется API-функция, предназначенная для вызова
         из обработчиков прерываний!!! */
      xQueueSendToFrontFromISR(xPrintQueue, &(pcStringsToPrint[2]),
            &xHigherPriorityTaskWoken);
      iCount = 0;
   }
}
/*-----------------------------------------------------------*/

static void prvPrintTask(void *pvParameters) {
   int iIndexToString;

   /* Будет создано 2 экземпляра этой задачи. В качестве параметра
      при создании задачи выступает номер строки в таблице строк. */
   iIndexToString = (int) pvParameters;

   for (;;) {
      /* Вывести строку на консоль. Но не напрямую, а передав
         указатель на строку задаче-сторожу.*/
      xQueueSendToBack(xPrintQueue, &(pcStringsToPrint[iIndexT
oString]), 0);

      /* Блокировать задачу на промежуток времени случайной
         длины: от 0 до 500 квантов. */
      vTaskDelay((rand() % 500));
      /* Вообще функция rand() не является реентерабельной.
       Однако в этой программе это неважно. */
   }
}
```

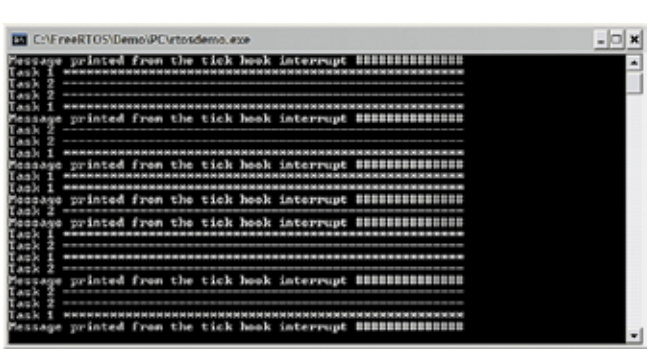

Рис. 9. Результат выполнения учебной программы № 3

Результат работы учебной программы № 3 приведен на рис. 9, на котором видно, что строки от двух задач с разными приоритетами и из тела обработчика прерывания выводятся на консоль без искажений. Следовательно, механизм взаимного исключения работает правильно.

Следует отметить, что в учебной программе № 3 приоритет задачи-сторожа задан самым низким в системе, поэтому строки накапливаются в очереди, пока задачи, их генерирующие, не заблокируются обе. В других ситуациях может потребоваться назначить задаче-сторожу более высокий приоритет. Это позволит ускорить прохождение очереди, но приведет к тому, что задача-сторож задержит выполнение более низкоприоритетных задач.

Выводы

В статье освещены вопросы организации совместного доступа к разделяемым ресурсам микроконтроллера. В дальнейших публикациях речь пойдет о сопрограммах — способе реализации многозадачной среды на микроконтроллерах с небольшим объемом оперативной памяти. Также внимание будет уделено нововведению версии FreeRTOS V7.0.0 — встроенной реализации программных таймеров.

Литература

1. Курниц А. FreeRTOS — операционная система для микроконтроллеров // Компоненты и технологии. 2011. № 2–7.
2. Barry R. Using the FreeRTOS real time kernel: A Practical Guide. 2009.
3. **http://www.freertos.org**
4. **http://ru.wikipedia.org/wiki/Реентерабельность**
5. **http://ru.wikipedia.org/wiki/Мьютекс**
6. **http://en.wikipedia.org/wiki/Reentrant_mutex**
7. **http://www.qnxclub.net/files/articles/invers/invers.pdf**
8. **http://ru.wikipedia.org/wiki/Взаимная_блокировка**
9. **http://www.ee.ic.ac.uk/t.clarke/rtos/lectures/RTOSlec2x2bw.pdf**

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.

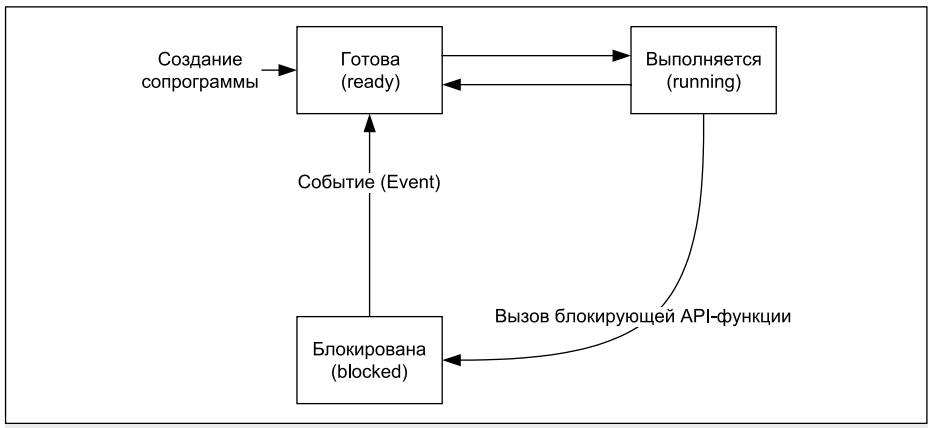

Рис. 1. Состояния сопрограммы

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

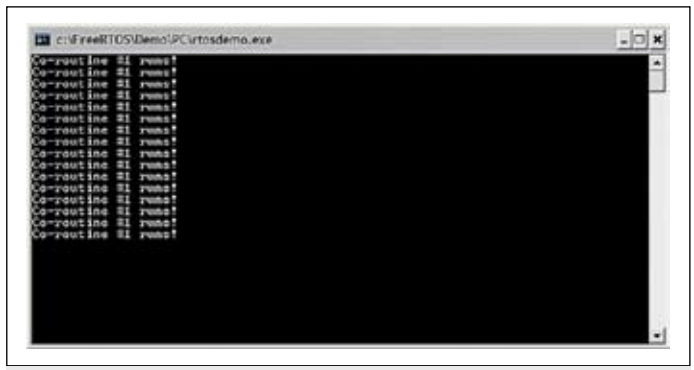

Рис. 4. Результат выполнения учебной программы № 1, когда Сопрограмма 1 не выполняет переключения на другую сопрограмму

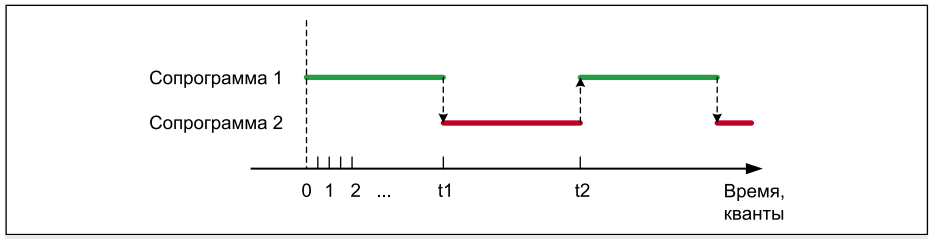

Рис. 3. Ход выполнения сопрограмм в учебной программе № 1

```
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.

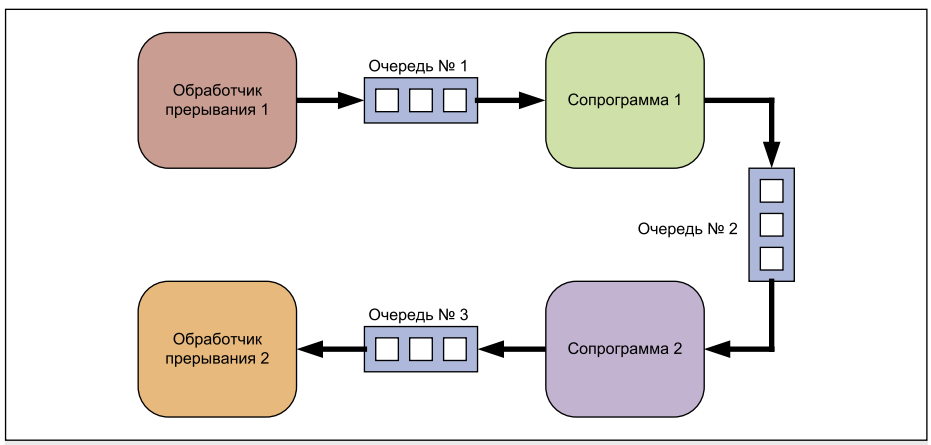

Рис. 5. Обмен сообщениями между сопрограммами и обработчиками прерываний в учебной программе № 2

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();
      }
}
```

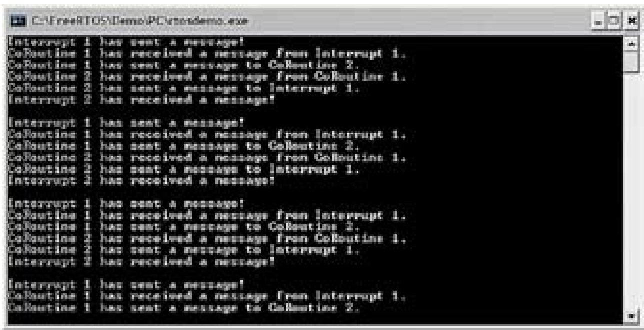

Рис. 6. Результат выполнения учебной программы № 2

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

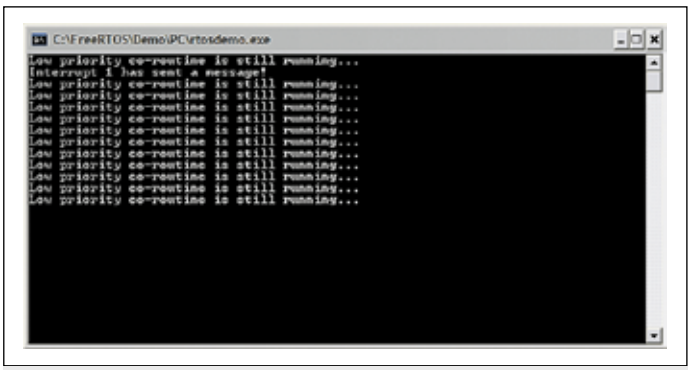

Рис. 7. Работа модифицированной учебной программы № 2

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

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.

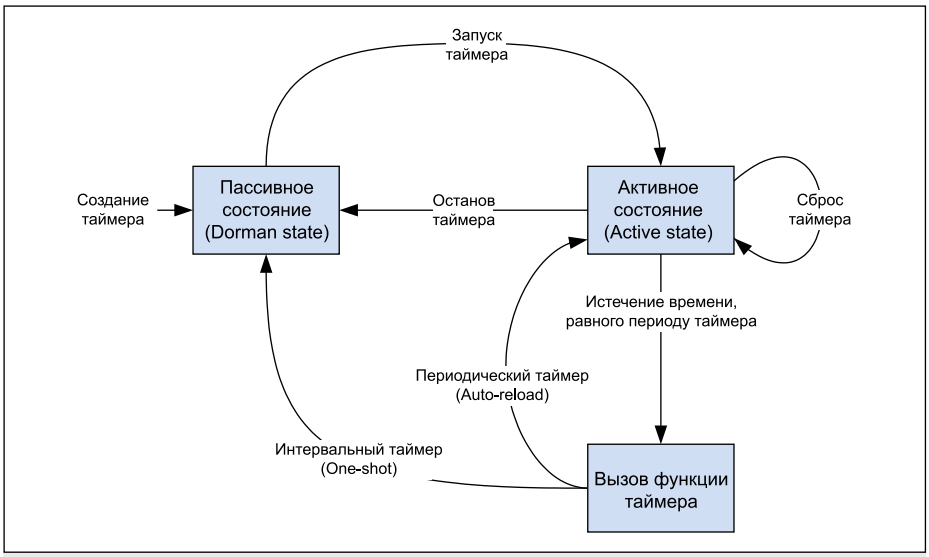

Рис. 1. Операции с таймером, состояния таймера и переходы между ними

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.

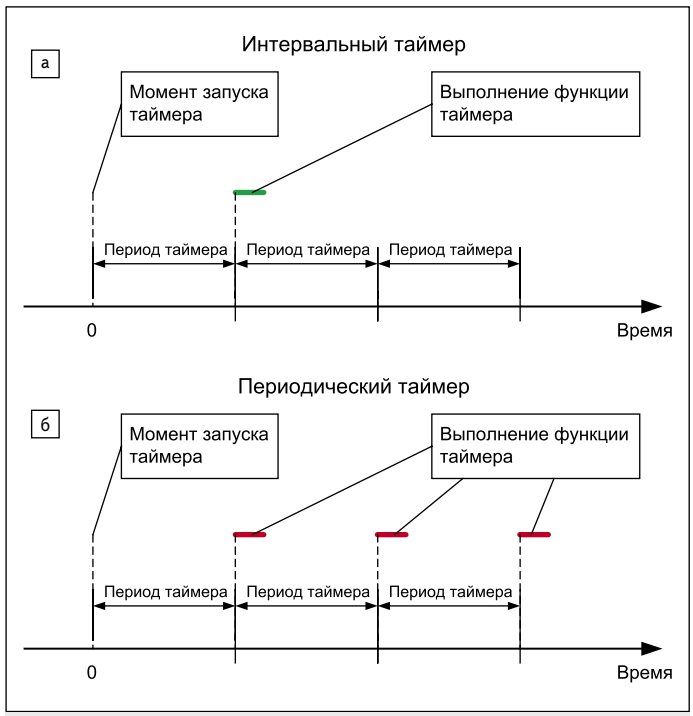

Рис. 2. Работа интервального и периодического таймера

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).

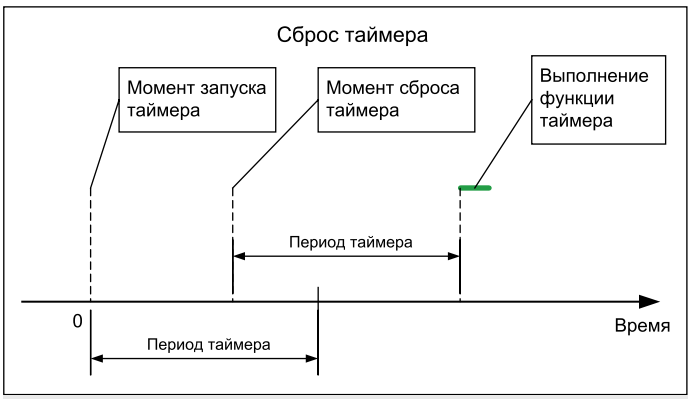

Рис. 3. Влияние сброса таймера на отсчет времени

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.

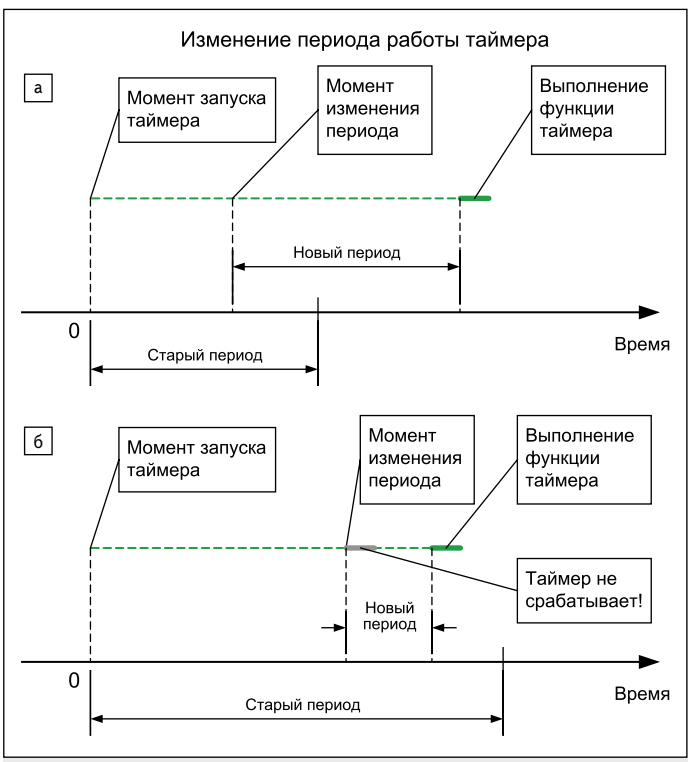

Рис. 4. Влияние изменения периода таймера на отсчет времени

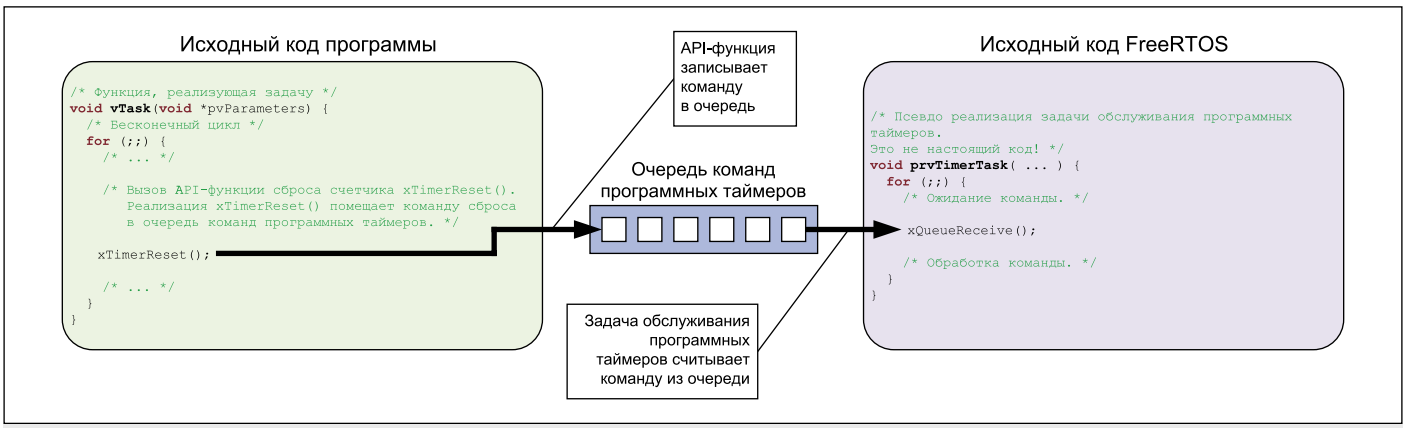

Рис. 5. Передача команды вследствие вызова API-функции сброса таймера

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);
   }
```

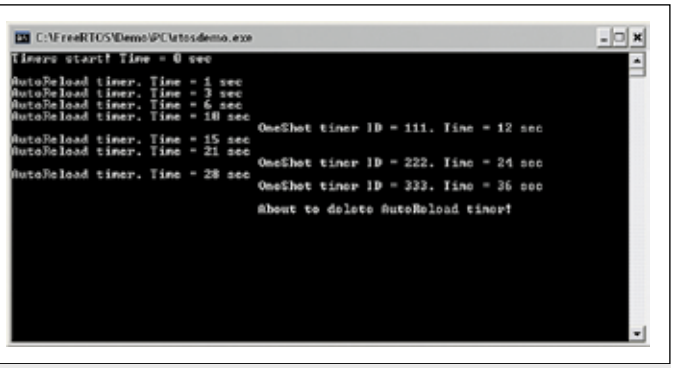

Рис. 6. Выполнение учебной программы

```
   /* 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.

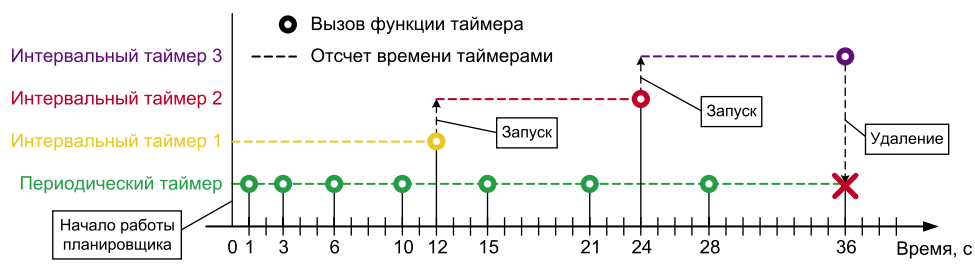

Рис. 7. Отсчет временных промежутков в учебной программе

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