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 внимание будет уделено этапу отладки приложения: мы рассмотрим возможности FreeRTOS, которые помогают найти трудно выявляемые ошибки в программе, определить узкие места программы и оценить пути ее дальнейшего расширения.

Возможности трассировки

Трассировка позволяет исследовать деятельность ядра FreeRTOS и проследить последовательность и время, когда задачи сменяют друг друга, а также отследить другие события ядра.

Последняя версия FreeRTOS (на момент написания статьи — V7.0.1) поддерживает два способа трассировки выполняющейся программы:

1. Механизм трассировки с записью в буфер (Legacy Trace Utility).
2. С использованием макросов трассировки (Trace Hook Macros).

Трассировка с записью в буфер

Такой метод трассировки заключается в том, что информация о последовательности выполнения задач во FreeRTOS и времени выполнения каждой задачи записывается в специально отведенный для этого буфер в оперативной памяти. Далее одну из задач можно запрограммировать так, чтобы она выводила содержимое буфера с трассировочной информацией через какой-либо интерфейс (RS-232C, например) или записывала в ПЗУ для дальнейшего изучения.

Чтобы возможность трассировки с записью в буфер была доступна, необходимо установить значение макроопределения ***configUSE_TRACE_FACILITY*** в файле ***FreeRTOSConfig.h*** равным 1. (Значение «0», соответственно, исключает механизм трассировки с записью в буфер из ядра FreeRTOS.)

Для того чтобы начать запись информации о работе ядра FreeRTOS, следует вызвать API-функцию ***vTaskStartTrace()***. Ее прототип:

```
void vTaskStartTrace( portCHAR * pcBuffer, unsigned portLONG ulBufferSize );
```

Аргументы:

1. ***pcBuffer*** — указатель на буфер, в который будет записана трассировочная информация.
2. ***ulBufferSize*** — задает предельный размер буфера. Когда в буфер будет записано ***ulBufferSize*** байт информации, трассировка будет автоматически остановлена.

Трассировка может быть остановлена принудительно, с помощью вызова API-функции ***ulTaskEndTrace()***. Запись трассировочной информации также останавливается, когда буфер будет заполнен. Прототип API-функции ***ulTaskEndTrace()***:

```
unsigned portLONG ulTaskEndTrace( void );
```

API-функция ***ulTaskEndTrace()*** возвращает фактическое количество байт трассировочной информации, которая была записана в буфер.

Информация в буфере записывается в бинарном виде. Утилита TraceCon позволяет конвертировать бинарное содержимое буфера в удобочитаемую текстовую форму. Утилита TraceCon входит в дистрибутив FreeRTOS (располагается в директории ***/TraceCon***) и выполняется в среде DOS или Windows.

FreeRTOS портирована на различные архитектуры, среди которых есть процессоры с различным порядком следования байтов [3]. Поэтому в директории ***/TraceCon*** находятся две версии утилиты TraceCon:

1. ***tracecon_little_endian.exe*** — предназначена для процессоров с порядком следования от младшего к старшему («интеловский» порядок байтов).
2. ***tracecon_big_endian_untested.exe*** — предназначена для процессоров с порядком следования от старшего к младшему («сетевой», или «мотороловский» порядок байтов).

Порядок работы с утилитой TraceCon следующий. Двоичное содержимое буфера с трассировочной информацией необходимо скопировать в файл ***TRACE.BIN***. Далее этот файл нужно поместить в одной директории вместе с утилитой TraceCon. После чего ее необходимо выполнить. В результате утилита создаст в текущей директории файл ***TRACE.TXT***, содержащий трассировочную информацию в текстовой форме.

Текстовый файл с трассировочной информацией содержит записи вида:

```
<Время>     <Номер задачи>
```

Пример фрагмента текстового трассировочного файла:

```
5012.450000  56
5012.500000  56
5012.500000  57
5012.550000  57
5012.550000  58
5012.600000  58
5012.600000  63
5013.000000  63
5013.000000  66
5013.050000  66
5013.050000  0
5013.100000  0
5013.100000  1
5013.150000  1
5013.150000  2
```

Графа «**Время**» определяет, сколько времени прошло с момента запуска планировщика, когда задача с номером («**Номер задачи**») получила управление.

Время в трассировочном файле представлено в системных квантах. Однако переключение контекста может происходить чаще, чем частота следования системных квантов (например, при блокировании задачи). Поэтому при переключении контекста в течение системного кванта время может быть записано не точно, но последовательность записей в трассировочном файле точно отражает последовательность выполнения задач в программе. Тем не менее для конкретного порта FreeRTOS точность представления времени в трассировочном файле может быть повышена за счет использования текущего значения таймера/счетчика, который отсчи-

тывает системные кванты времени. В приведенном примере точность представления времени составляет 0,05 кванта времени.

Второй графой в трассировочном файле выступает номер задачи. Он уникален для каждого экземпляра каждой задачи. Номер задачи является составной частью блока управления задачей (Task Control Block) и используется только для трассировки. Если трассировка с записью в буфер отключена (макроопределение ***configUSE_TRACE_FACILITY*** равно «0»), то номер задачи автоматически исключается из блока управления задачей.

Получить номер задачи можно при помощи API-функции ***vTaskList()***, которая выводит список всех задач в программе. Когда выполняется API-функция ***vTaskList()***, прерывания процессора запрещены, то есть программа находится в критической секции. Отлаженная программа не должна содержать вызовов ***vTaskList()***, применять ее следует только для отладки.

Пример списка задач, полученный с помощью вызова API-функции ***vTaskList()***:

```
Name          State  Priority  Stack  Num
*********************************************
Print         R      4         358    64
QConsB6       R      0         192    58
Rec3          R      0         176    63
C_CTRL        R      0         193    31
IDLE          R      0         212    66
Math1         R      0         442    0
QConsNB       B      2         190    11
BTest1        B      7         195    19
FBTest1       B      3         193    51
QConsNB       B      2         192    59
QProdNB       B      2         190    12
QProdNB       B      2         192    60
PeekM         S      1         204    42
Event2        S      3         194    39
MuHigh        S      3         196    24
MuMed         S      2         218    23
Event3        S      3         194    40
FMuHigh       S      3         206    50
FMuMed        S      2         218    49
COMRx         S      3         185    9
```

Список задач содержит следующие поля:

1. ***Name***. Имя задачи, которое было назначено ей в момент создания (второй аргумент API-функции ***xTaskCreate()***).
2. ***State***. Состояние задачи в момент вызова ***vTaskList()***. Может принимать следующие значения:
   - «R» — Ready, состояние готовности к выполнению.
   - «B» — Blocked, блокированное состояние задачи.
   - «S» — Suspended, приостановленное состояние.
   - «D» — Deleted. Означает, что задача была удалена.
3. ***Priority***. Приоритет задачи в момент вызова ***vTaskList()***.
4. ***Stack***. Максимальный объем стека, который использовала задача за время своей работы.
5. ***Num***. Уникальный номер задачи в системе. Так как может быть создано несколько экземпляров одной задачи, то поле ***Name*** может повторяться в списке (***QProdNB*** в примере выше), однако номер задачи уникален для каждого ее экземпляра.

Следует отметить, что поле ***State*** не индицирует состояние выполнения задачи: задача, которая в данный момент выполняется, будет отображаться как задача, находящаяся в состоянии готовности к выполнению.

Также следует обратить внимание на поле ***Stack***. Оно отображает не размер стека, который был выделен задаче при ее создании, а фактически использованный объем памяти стека. Причем запоминается максимальная величина этого объема за все время существования задачи. Поле ***Stack*** удобно использовать для экспериментального нахождения размера стека, который необходимо выделить задаче при ее создании.

Как и при выполнении API-функции ***vTaskStartTrace()***, список задач помещается в предварительно подготовленный буфер в оперативной памяти. Список в буфере представлен непосредственно в текстовой форме, как и показано в примере выше.

Прототип ***vTaskList()***:

```
void vTaskList( portCHAR *pcWriteBuffer );
```

Единственный аргумент ***pcWriteBuffer*** — указатель на буфер, в который будет помещен список задач в текстовом виде. Предполагается, что размер буфера достаточен для размещения всего списка. Необходимый размер буфера можно вычислить приблизительно, приняв, что для представления информации об одной задаче требуется около 40 байт.

Чтобы API-функция ***vTaskList()*** была доступна, необходимо, чтобы макроопределения ***configUSE_TRACE_FACILITY***, ***INCLUDE_vTaskDelete*** и ***INCLUDE_vTaskSuspend*** в файле ***FreeRTOSConfig.h*** были равны 1.

Оценить возможности трассировки с записью в буфер можно на примере демонстрационного проекта для порта FreeRTOS для x86-процессора, работающего в реальном режиме (именно этот порт использовался в большинстве учебных программ в предыдущих публикациях [1]).

Возможность трассировки уже включена в демонстрационный проект, однако, чтобы выполнить трассировку выполняющейся программы, нужно сделать несколько изменений в проекте.

Прежде всего необходимо задать место расположения трассировочного файла и файла со списком задач, так как изначально их место расположения задано как корневая директория диска A:. Сделать это можно, отредактировав файл ***fileIO.c***, находящийся в директории ***\Demo\PC\FileIO***. Строку:

```
const char * const pcFileName = "a:\\RTOSlog.txt";
```

необходимо заменить на:

```
const char * const pcFileName = "c:\\RTOSlog.txt";
```

а строку:

```
const char * const pcFileName = "a:\\trace.bin";
```

следует заменить на строку:

```
const char * const pcFileName = "c:\\trace.bin";
```

Таким образом, трассировочный файл и файл со списком задач будут созданы в корневой директории диска C:\. Если нужно выбрать другое место расположения файлов, то нужно задать соответствующий путь в строках, приведенных выше.

Далее следует выполнить сборку демонстрационного проекта в среде Open Watcom IDE. Как установить и настроить эту среду, рассказано в [1, № 4]. Обязательным условием для того, чтобы трассировка работала, является сборка демонстрационного проекта в Release-конфигурации. Однако изначально выбрана отладочная Debug-конфигурация, в которой возможность трассировки отключена.

Для того чтобы переключить конфигурацию проекта на Release, необходимо в среде Open Watcom IDE выбрать пункт меню ***Targets → Target Options → Use Release Switches*** (рис. 1).

Кроме того, необходимо убедиться, что возможность трассировки включена в конфигурационном файле ***FreeRTOSConfig.h***, который должен содержать строку:

```
#define configUSE_TRACE_FACILITY       1
```

Когда все вышеперечисленные условия выполнены, можно выполнить сборку демонстрационного проекта и запустить полученный исполняемый ****.exe*** файл на выполнение.

Получить файл со списком задач можно, если во время выполнения демонстрационного проекта нажать «t» на клавиатуре. При

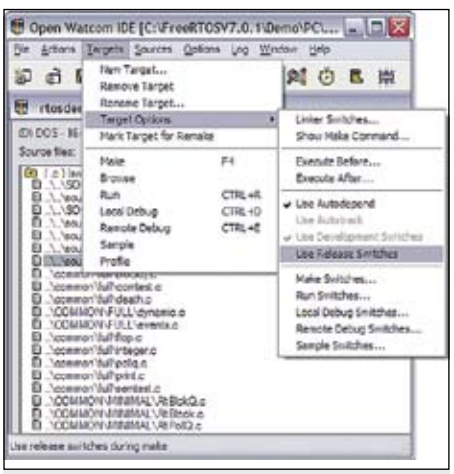

Рис. 1. Выбор Release-конфигурации

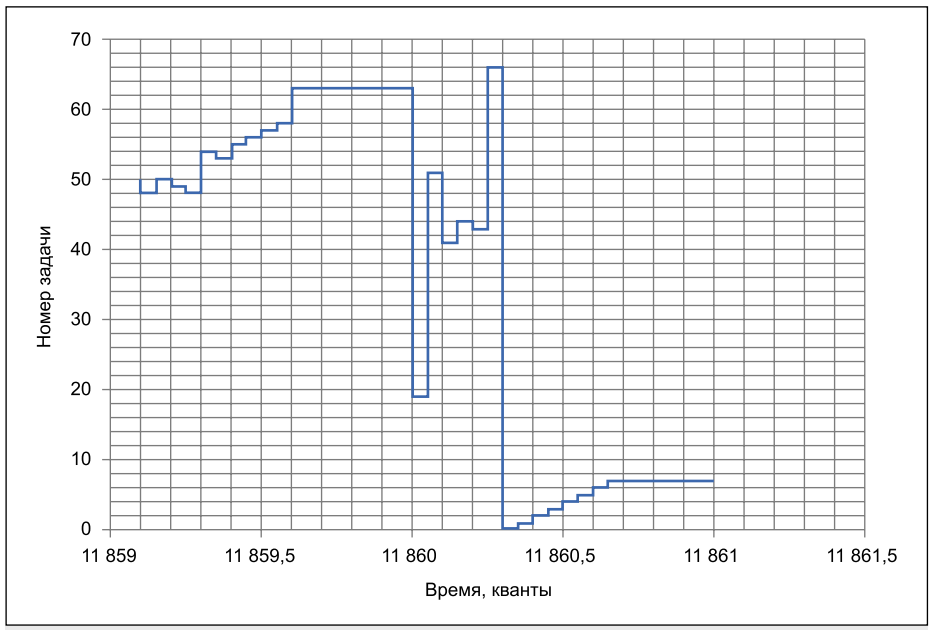

Рис. 2. Графическое представление отладочной информации

этом в корневой директории диска C: появится файл ***RTOSlog.txt,*** содержащий список всех задач в системе.

Начать трассировку можно, нажав «s» на клавиатуре, а чтобы закончить — клавишу «e». При этом в корневой директории диска C: появится трассировочный файл ***Trace.bin***, содержащий трассировочную информацию в бинарном виде.

Для того чтобы получить текстовый файл с трассировочной информацией, необходимо скопировать полученный файл ***Trace.bin*** в одну директорию с утилитой ***tracecon_little_endian.exe*** и запустить ее. В результате в этой же директории будет создан готовый для анализа текстовый трассировочный файл ***Trace.txt***.

Графическое представление трассировочной информации

Изучать поведение программы, просматривая содержимое текстового трассировочного файла, может быть трудно. Гораздо удобнее графическое представление трассировочной информации. При этом переключения между задачами отображаются в виде двухмерного графика, по оси абсцисс которого откладывается время, а по оси ординат — номер выполняемой в данный момент задачи.

Получить графическое представление информации из текстового трассировочного файла можно различными способами. Один из них, который будет рассмотрен ниже, — с помощью редактора электронных таблиц Microsoft Excel. Рассматривается работа с русской версией Microsoft Excel 2007/2010.

В главном меню программы необходимо выбрать закладку «**Данные**», далее «**Получить внешние данные**» → «**Из текста**». После этого «мастер» предложит выбрать файл — источник данных, в качестве которого следует выбрать полученный ранее текстовый трассировочный файл. После импортирования данных первый столбец таблицы должен содержать числовое значение времени, а второй — номера задачи. Каждая строка таблицы будет соответствовать факту переключения процессора на другую задачу.

Далее следует выделить интересующий участок полученной таблицы и выбрать вкладку «**Вставка**», затем «**Диаграммы**» → «**Точечная**» → «**Точечная с прямыми отрезками**». В результате мы получим график, подобный представленному на рис. 2.

Трассировка с помощью макросов трассировки

Начиная с версии V4.8.0 во FreeRTOS появился более универсальный, чем трассировка с записью в буфер, механизм трассировки — макросы. Макросы трассировки — это инструмент, позволяющий получать расширенную информацию о поведении программы, работающей под управлением FreeRTOS.

Макросы трассировки дают возможность:

1. Записать последовательность выполнения задач.
2. Измерить время выполнения каждой задачи.
3. Зафиксировать факт наступления и время, когда происходят определенные события ядра или вызовы API-функций.
4. Получать информацию только об определенных событиях, связанных с определенными задачами или очередями, и оставить без внимания остальную активность ядра.

Главная «изюминка» реализации макросов трассировки — это то, что исходный код самой FreeRTOS содержит вызовы изначально пустых макросов, которые включены во все основные операции внутри ядра: переключение контекста, вызов определенной API-функции и т. д. Макросы трассировки могут быть переопределены в программе для выполнения любых предусмотренных программистом действий. Обычно эти действия направлены на какую-либо индикацию, отображение и, возможно, запись факта вызова того или иного макроса трассировки.

Совершенно необязательно задавать полезные действия для всех трассировочных макросов, достаточно переопределить те, которые автоматически будут вызваны в результате интересующих событий (например, при блокировке задачи, переключении контекста и т. п.). Непереопределенные макросы останутся пустыми и не будут оказывать влияние на временные характеристики программы.

Приведем несколько примеров, как можно реализовать индикацию переключения задач в макросах трассировки:

1. Использование нескольких цифровых выходов микроконтроллера (МК) для индикации номера выполняемой в данный момент задачи. Номер задачи можно задавать, например, в двоичном или позиционном коде. К этим цифровым выходам может быть подключен цифровой анализатор для записи и дальнейшего анализа последовательности выполнения задач.
2. Использование одного аналогового выхода, при этом аналоговый уровень напряжения будет индицировать выполняющуюся в данный момент задачу. Этот способ подобен первому, но вместо цифрового анализатора для записи трассировочной информации потребуется обычный осциллограф.

Макросы трассировки можно использовать для интеграции с отладчиками сторонних производителей.

Тэг задачи

Тэг задачи подобен рассмотренному выше номеру задачи и представляет собой дополнительный параметр, который включается в состав блока управления задачей, если макроопределение ***configUSE_APPLICATION_TASK_TAG*** равно 1.

Тэг задачи определен как переменная типа ***pdTASK_HOOK_CODE***. Тип ***pdTASK_HOOK_CODE*** определен как указатель на функцию:

```
typedef portBASE_TYPE (*pdTASK_HOOK_CODE)( void * );
```

Однако в программе тэг задачи можно использовать и как указатель на произвольный тип данных, и как целочисленное значение.

Для того чтобы назначить задаче определенный тэг, служит API-функция ***vTaskSetApplicationTaskTag()***. Ее прототип:

```
void vTaskSetApplicationTaskTag( xTaskHandle xTask, pdTASK_
HOOK_CODE pxTagValue );
```

Аргумент ***xTask*** служит для задания дескриптора задачи, тэг которой необходимо задать. Аргумент ***pxTagValue*** непосредственно определяет само значение тэга.

Тэг задачи можно использовать для того, чтобы пометить (пронумеровать) те задачи в программе, поведение которых необходимо изучить. Далее в макросах трассировки можно выводить тэг задачи на аналоговый или цифровой(ые) выводы МК. Таким образом, можно получить индикацию выполнения только определенного набора задач.

Рассмотрим пример использования тэга задачи в макросах трассировки. В программе двум задачам будут назначены тэги, равные, соответственно, 1 и 2. Далее будет переопределен трассировочный макрос ***traceSWITCHED_IN()*** (вызывается каждый раз при переключении на следующую задачу) так, чтобы он выводил значение тэга текущей задачи на вывод микроконтроллера:

```
/* Функция, реализующая Задачу 1 */
void vTask1( void *pvParameters )
  {
  /* Задать для этой задачи тэг, равный 1 */
vTaskSetApplicationTaskTag( NULL, ( void * ) 1 );

  for( ;; )
    {
    /* Полезный код */
    }
  }
/*-----------------------------------------------------*/

/* Функция, реализующая Задачу 2 */
void vTask1( void *pvParameters )
  {
  /* Задать для этой задачи тэг, равный 2 */
  vTaskSetApplicationTaskTag( NULL, ( void * ) 2 );

  for( ;; )
    {
    /* Полезный код */
    }
  }
/*-----------------------------------------------------*/

/* Переопределить трассировочный макрос
traceTASK_SWITCHED_IN() так, чтобы он устанавливал
аналоговое напряжение на выводе пропорционально тэгу
задачи, которая вызвала этот макрос. */
#define traceTASK_SWITCHED_IN() vSetAnalogueOutput( 0, ( int )
pxCurrentTCB->pxTaskTag )
```

Помимо фиксации факта переключения задачи с одной на другую, макросы трассировки позволяют зарегистрировать причину, по которой переключение контекста имело место. Рассмотрим пример:

```
/* traceBLOCKING_ON_QUEUE_RECEIVE() — один из макросов,
позволяющий определить причину переключения
с одной задачи на другую. */
#define traceBLOCKING_ON_QUEUE_RECEIVE(pxQueue)        \
      ulSwitchReason = reasonBLOCKING_ON_QUEUE_READ;

/* log_event() — это функция, определенная программистом,
которая получает через свои аргументы задачу, которая вышла
из состояния выполнения, и причину, почему это произошло. */
#define traceTASK_SWITCHED_OUT()                       \
      log_event( pxCurrentTCB, ulSwitchReason );
```

Макросы, которые вызываются из прерываний, а в частности — из прерывания таймера/счетчика, который отсчитывает системные кванты времени, должны выполняться как можно быстрее. Операции «установка значений переменных», «запись в порт ввода/вывода или регистр специальных функций МК» допустимы, напротив, попытка записи отладочного сообщения с помощью ***fprintf()*** не будет работать.

Таблица. Список макросов трассировки

Идентификатор макроса трассировки	Описание
traceTASK_INCREMENT_TICK(xTickCount)	Вызывается из тела прерывания таймера/счетчика, который отсчитывает системные кванты времени. Параметр: текущее значение счетчика системных квантов
traceTASK_SWITCHED_OUT()	Вызывается перед тем, как задача выйдет из состояния выполнения и управление получит другая задача. При этом глобальная переменная pxCurrentTCB содержит дескриптор задачи, которую вытеснит другая задача
traceTASK_SWITCHED_IN()	Вызывается в момент, когда задача переходит в состояние выполнения. При этом глобальная переменная pxCurrentTCB содержит дескриптор этой задачи
traceBLOCKING_ON_QUEUE_RECEIVE(pxQueue)	Индицирует, что текущая задача переходит в блокированное состояние, когда она сделала попытку: — прочитать из пустой очереди; — захватить уже захваченный семафор или мьютекс. Параметр: дескриптор очереди, семафора или мьютекса
traceBLOCKING_ON_QUEUE_SEND(pxQueue)	Индицирует, что текущая задача переходит в блокированное состояние, когда она сделала попытку записи в полностью заполненную очередь. Параметр: дескриптор очереди
traceGIVE_MUTEX_RECURSIVE(pxMutex)	Вызывается, когда API-функция xSemaphoreGiveRecursive() пытается освободить мьютекс. Параметр: дескриптор мьютекса
traceGIVE_MUTEX_RECURSIVE_FAILED(pxMutex)	Вызывается, когда попытка освободить мьютекс с помощью API-функции xSemaphoreGiveRecursive() оказывается неуспешной. Параметр: дескриптор мьютекса
traceQUEUE_CREATE(pxNewQueue)	Вызывается из API-функции xQueueCreate(), если очередь успешно создана. Параметр: дескриптор очереди
traceQUEUE_CREATE_FAILED()	Вызывается из API-функции xQueueCreate(), если очередь не создана из-за отсутствия достаточного объема памяти. Параметр: дескриптор очереди
traceCREATE_MUTEX(pxNewMutex)	Вызывается из API-функции xSemaphoreCreateMutex(), если мьютекс успешно создан. Параметр: дескриптор мьютекса
traceCREATE_MUTEX_FAILED()	Вызывается из API-функции xSemaphoreCreateMutex(), если мьютекс не создан (мало памяти)
traceGIVE_MUTEX_RECURSIVE(pxMutex)	Вызывается из API-функции xSemaphoreGiveRecursive(), если мьютекс успешно возвращен. Параметр: дескриптор мьютекса
traceGIVE_MUTEX_RECURSIVE_FAILED(pxMutex)	Вызывается из API-функции xSemaphoreGiveRecursive(), если мьютекс не возвращен из-за того, что вызывающая задача не захватила мьютекс ранее (не является его владельцем). Параметр: дескриптор мьютекса
traceTAKE_MUTEX_RECURSIVE(pxMutex)	Вызывается из API-функции xQueueTakeMutexRecursive(). Параметр: дескриптор мьютекса
traceCREATE_COUNTING_SEMAPHORE()	Вызывается из API-функции xSemaphoreCreateCounting(), если семафор успешно создан
traceCREATE_COUNTING_SEMAPHORE_FAILED()	Вызывается из API-функции xSemaphoreCreateCounting(), если семафор не создан (мало памяти)
traceQUEUE_SEND(pxQueue)	Вызывается из API-функций xQueueSend(), xQueueSendToFront(), xQueueSendToBack() или из любой API-функции, которая возвращает семафор, если операция записи в очередь была успешно выполнена. Параметр: дескриптор очереди или семафора
traceQUEUE_SEND_FAILED(pxQueue)	Вызывается из API-функций xQueueSend(), xQueueSendToFront(), xQueueSendToBack() или из любой API-функции, которая возвращает семафор, если операция записи в очередь не была выполнена, так как очередь оставалась заполненной на протяжении указанного времени блокировки. Параметр: дескриптор очереди или семафора
traceQUEUE_RECEIVE(pxQueue)	Вызывается из API-функции xQueueReceive() или из любой API-функции, которая захватывает семафор, если операция чтения из очереди выполнена успешно. Параметр: дескриптор очереди или семафора
traceQUEUE_RECEIVE_FAILED(pxQueue)	Вызывается из API-функции xQueueReceive() или из любой API-функции, которая захватывает семафор, если операция чтения из очереди не была выполнена, так как очередь оставалась пустой на протяжении указанного времени блокировки. Параметр: дескриптор очереди или семафора
traceQUEUE_PEEK(pxQueue)	Вызывается из API-функции xQueuePeek(). Параметр: дескриптор очереди
traceQUEUE_SEND_FROM_ISR(pxQueue)	Вызывается из API-функции xQueueSendFromISR(), если запись в очередь прошла успешно. Параметр: дескриптор очереди
traceQUEUE_SEND_FROM_ISR_FAILED(pxQueue)	Вызывается из API-функции xQueueSendFromISR(), если запись в очередь не произошла, так как очередь заполнена. Параметр: дескриптор очереди
traceQUEUE_RECEIVE_FROM_ISR(pxQueue)	Вызывается из API-функции xQueueReceiveFromISR(), если чтение из очереди прошло успешно. Параметр: дескриптор очереди
traceQUEUE_RECEIVE_FROM_ISR_FAILED(pxQueue)	Вызывается из API-функции xQueueReceiveFromISR(), если чтение из очереди не произошло, так как очередь пуста. Параметр: дескриптор очереди
traceQUEUE_DELETE(pxQueue)	Вызывается из API-функции vQueueDelete(). Параметр: дескриптор удаляемой очереди
traceTASK_CREATE(pxTask)	Вызывается из API-функции xTaskCreate(), если задача успешно создана. Параметр: дескриптор создаваемой задачи
traceTASK_CREATE_FAILED(pxNewTCB)	Вызывается из API-функции xTaskCreate(), если задача не создана из-за отсутствия памяти. Параметр: указатель на блок управления задачей, для размещения которого не хватило памяти
traceTASK_DELETE(pxTask)	Вызывается из API-функции vTaskDelete(). Параметр: дескриптор удаляемой задачи
traceTASK_DELAY_UNTIL()	Вызывается из API-функции vTaskDelayUntil()
traceTASK_DELAY()	Вызывается из API-функции vTaskDelay()
traceTASK_PRIORITY_SET(pxTask,uxNewPriority)	Вызывается из API-функции vTaskPrioritySet(). Параметры: 1) Дескриптор задачи. 2) Новое значение приоритета
traceTASK_SUSPEND(pxTask)	Вызывается из API-функции vTaskSuspend(). Параметр: дескриптор приостанавливаемой задачи
traceTASK_RESUME(pxTask)	Вызывается из API-функции vTaskResume(). Параметр: дескриптор задачи
traceTASK_RESUME_FROM_ISR(pxTask)	Вызывается из API-функции xTaskResumeFromISR(). Параметр: дескриптор задачи

Переопределения макросов необходимо сделать в программе до включения заголовочного файла ***FreeRTOS.h***. Наиболее простой путь добиться этого — разместить переопределения макросов трассировки в конце файла ***FreeRTOSConfig.h***. Можно также создать отдельный заголовочный файл с трассировочными макросами и включить его в конец файла ***FreeRTOSConfig.h***.

Трассировочный макрос может иметь параметр, который определяет, какая задача, очередь, семафор или мьютекс связаны с событием, в результате которого был вызван данный макрос. Полный список макросов трассировки приведен в таблице.

Рассмотрим пример практического применения макросов трассировки. В качестве аппаратной платформы будем использовать микро-

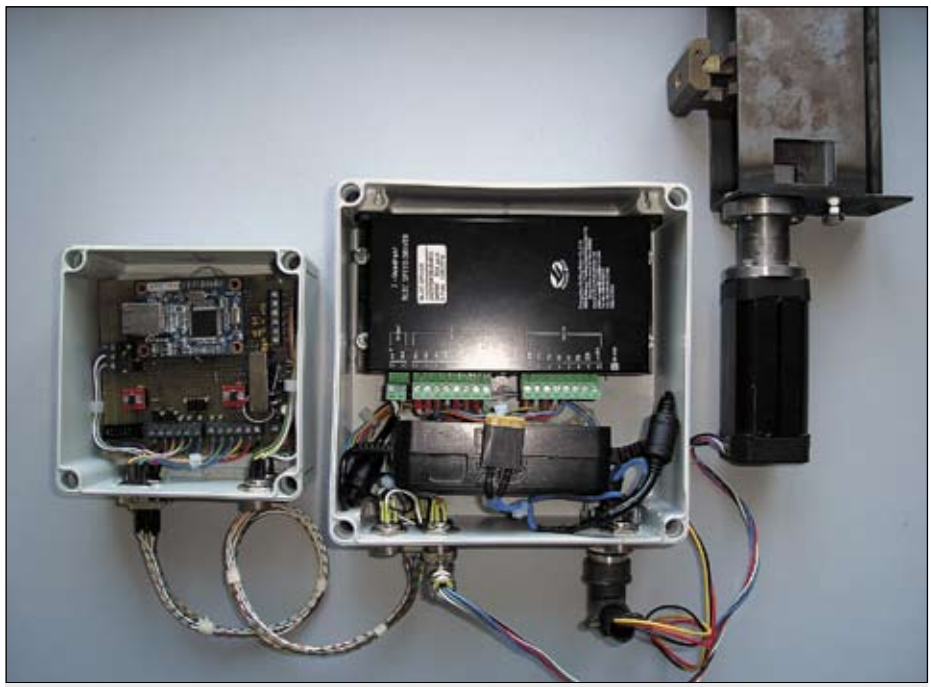

Рис. 3. Опытный образец контроллера бесколлекторного двигателя постоянного тока

контроллер AVR ATmega128L, установленный на мезонинный модуль WIZ200WEB фирмы WIZnet. Установка и настройка FreeRTOS для этой платформы рассматривались ранее [1, № 3], поэтому приводить здесь их не будем.

AVR ATmega128L является основой контроллера бесколлекторного двигателя постоянного тока (рис. 3).

Отлаживаемое приложение — непосредственно программа управления контроллером, выполняющаяся под управлением FreeRTOS.

Информация о положении вала двигателя поступает от трехфазного датчика Холла (энкодера), встроенного в двигатель. Каждая фаза энкодера подключена к выводу ATmega128L так, что перепад логического уровня на каждой фазе вызывает внешнее прерывание микроконтроллера.

Программа управления построена так, что обработчик каждого внешнего прерывания от энкодера записывает приращение угла поворота вала в очередь ***xDeltaPositionQueue*** длиной в 10 элементов (рис. 4). Задача ***Servo***, занимающаяся непосредственно управлением двигателем, принимает приращения угла из очереди, складывает их и формирует, таким образом, текущий угол поворота вала двигателя.

Задача трассировки: выяснить, достаточна ли длина очереди в 10 элементов для того, чтобы при максимальной скорости вращения двигателя (когда внешние прерывания от энкодера происходят с максимальной частотой) все приращения угла от каждого прерывания были записаны в очередь и в дальнейшем обработаны задачей ***Servo***.

Для этого необходимо отследить событие неудавшейся попытки записи в очередь из тела обработчика прерывания. Трассировочный макрос ***traceQUEUE_SEND_FROM_ISR_FAILED(pxQueue)*** вызывается именно в таком случае. Поэтому переопределим его, добавив следующие строки в конец файла ***FreeRTOSConfig.h***:

```
#define traceQUEUE_SEND_FROM_ISR_FAILED(pxQueue)        \
   /* Если запись не удалась в очередь xDeltaPositionQueue. */  \
   if (pxQueue == xDeltaPositionQueue) {                \
      /* Инвертировать вывод общего назначения PG3. */  \
         DDRG   |= (1 << DDG3);                         \
         PORTG  ^= (1 << PG3);                          \
   }
```

В случае неудавшейся записи в очередь ***xDeltaPositionQueue*** будет инвертирован свободный вывод МК, состояние которого легко отследить с помощью осциллографа или даже по миганию светодиода. (К выводу PG3 подключен светодиод, расположенный прямо на модуле WIZ200WEB.)

Следует отметить, что в приведенном примере производится операция чтение/модификация/запись регистров DDRG и PORTG, что является одним из случаев совместного доступа к ресурсу, который требует применения одного из механизмов взаимного исключения [1, № 8]. Однако конкретно в этом случае макрос ***traceQUEUE_SEND_FROM_ISR_FAILED()*** вызывается из обработчика прерывания, а особенность архитектуры AVR в том, что прерывания по умолчанию не являются вложенными. Поэтому переключение контекста не может произойти во время выполнения этого макроса трассировки. Следовательно, в применении каких-либо механизмов взаимного исключения нет необходимости.

Выполним сборку проекта, когда длина очереди задана 10 элементов, и запустим полученную отладочную версию программы в МК. Состояние вывода PG3 будем отслеживать осциллографом. Можно наблюдать, что логический уровень на выводе PG3 не меняется (рис. 5). Следовательно, длины очереди

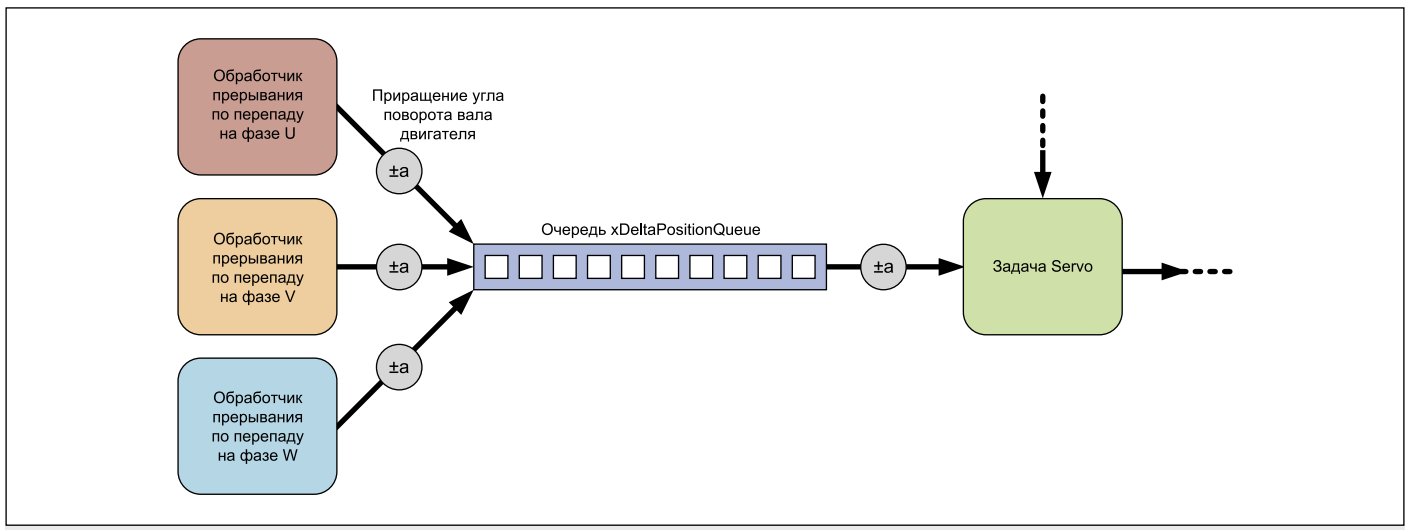

Рис. 4. Структурная схема программы контроллера бесколлекторного двигателя постоянного тока (часть)

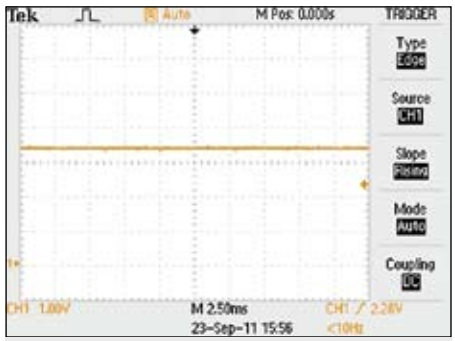

Рис. 5. Состояние вывода PG3. Все попытки записи в очередь успешны

xDeltaPositionQueue в 10 элементов достаточно для фиксации всех быстро следующих друг за другом событий (сигналов датчика Холла).

Если же выполнить сборку проекта, установив длину очереди в 3 элемента, то можно видеть (рис. 6), что логический уровень на выводе PG3 теперь инвертируется. Следовательно, длины очереди в 3 элемента оказалось недостаточно.

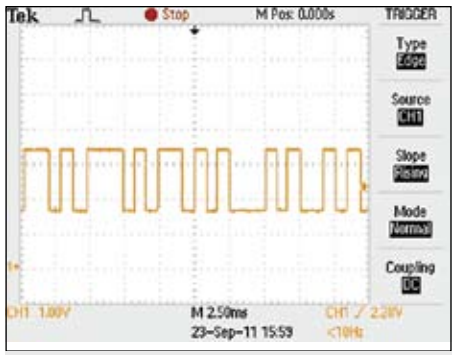

Рис. 6. Состояние вывода PG3. Ситуация, когда запись в очередь невозможна, наступает периодически

Получение статистики выполнения задач

Существует возможность настроить ядро FreeRTOS так, чтобы накапливалась статистика, сколько процессорного времени потребляет каждая из задач. Наряду с трассировкой статистика выполнения задач может оказаться полезной для правильного назначения приоритетов задачам, для определения возможности добавления дополнительных задач в программу, степени загрузки процессора и т. д.

API-функция ***vTaskGetRunTimeStats()*** служит для получения статистики в текстовой табличной форме в виде, представленном на рис. 7.

В данном случае для просмотра статистики используется интернет-браузер Google Chrome. Компьютер подключен по Ethernet-протоколу к целевому устройству, на котором выполняется встроенный веб-сервер. Выполняющаяся на МК программа вызывает API-функцию ***vTaskGetRunTimeStats()***, а результат ее работы представляет в виде интернет-страницы. Вывод статистики таким способом реализован в демонстрационных проектах для микроконтроллеров NXP LPC17xx Cortex M3 и TI Stellaris LM3Sxxxx из дистрибутива FreeRTOS.

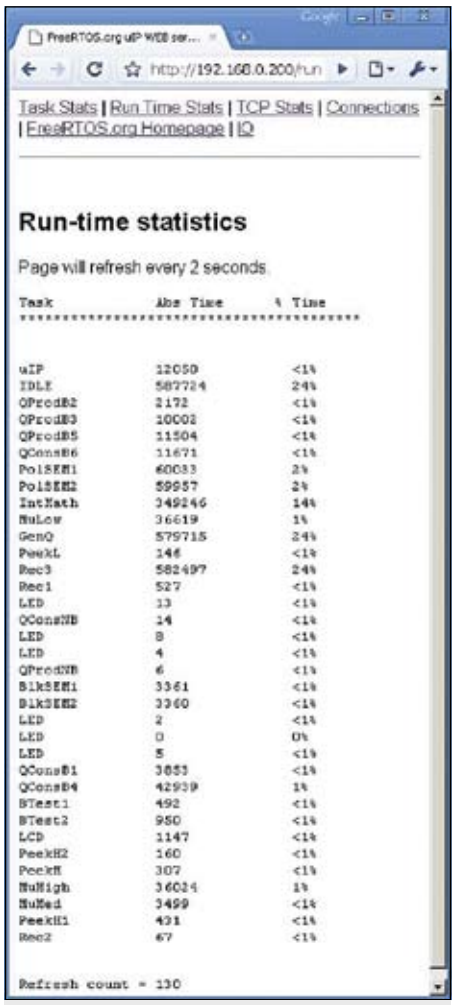

Рис. 7. Пример вывода статистики выполнения задач

В таблице статистики (рис. 7) каждой задаче (задается по имени, поэтому разные экземпляры одной задачи отображаются под одним именем) ставится в соответствие два параметра:

1. Абсолютное время выполнения (Abs time) — показывает, сколько единиц времени в сумме выполнялась та или иная задача до момента вызова ***vTaskGetRunTimeStats()***. Подробнее о единице времени сказано ниже.
2. Относительное время выполнения (% Time) — показывает, какую часть от общего объема времени (от момента запуска планировщика до вызова ***vTaskGetRunTimeStats()***) составляет суммарное время выполнения той или иной задачи. Представлено в процентах.

Судить о загруженности процессора можно по суммарному времени выполнения задачи Бездействие (IDLE). Чем больше загружен процессор, тем меньше времени в процентном отношении выполняется задача Бездействие.

Остановимся подробнее на единицах времени, в которых представлено абсолютное время выполнения задач. Величину этой единицы программист может устанавливать произвольно, для сохранения точности представления статистики следует устанавливать ее в пределах от 10 до 100 раз меньше одного системного кванта времени.

Сразу стоит оговориться, что для отсчета суммарного времени выполнения задач потребуется точный источник частоты, которая в 10–100 раз выше частоты следования системных квантов. В качестве такого источника целесообразно применять незадействованный таймер/счетчик микроконтроллера.

Настройки FreeRTOS для получения статистики выполнения задач:

1. Макроопределение ***configGENERATE_RUN_TIME_STATS*** в файле ***FreeRTOSConfig.h*** должно быть равно 1.
2. В программе необходимо определить макрос ***portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()***, который должен производить настройки таймера/счетчика для отсчета времени выполнения задач. Если макроопределение ***configGENERATE_RUN_TIME_STATS*** задано равным 1, то данный макрос автоматически вызывается при запуске планировщика API-функцией ***vTaskStartScheduler()***.
3. Также необходимо определить макрос ***portGET_RUN_TIME_COUNTER_VALUE()***, который должен возвращать текущее значение таймера/счетчика, используемого для отсчета единиц времени работы задач. Если макроопределение ***configGENERATE_RUN_TIME_STATS*** задано равным 1, то данный макрос автоматически вызывается при переключении контекста.

Остановимся подробнее на API-функции ***vTaskGetRunTimeStats()***. Следует отметить, что она запрещает прерывания МК на время своего выполнения. Поэтому ее нужно использовать только для отладки; релиз программы не должен содержать ее вызовов. Прототип ***vTaskGetRunTimeStats()***:

```
void vTaskGetRunTimeStats( portCHAR *pcWriteBuffer );
```

Единственный аргумент — указатель на буфер, в который будет помещена информация в текстовом виде. Размер буфера не задан; выделяя для него память, следует придерживаться того, что для записи информации об одной строке (об одной задаче) потребуется около 40 байт памяти.

Контроль переполнения стека

Как говорилось ранее [1, № 3], каждая задача во FreeRTOS имеет свой собственный стек. Память для размещения стека задачи выделяется на этапе создания задачи; объем этой памяти задается как параметр API-функции ***xTaskCreate()***.

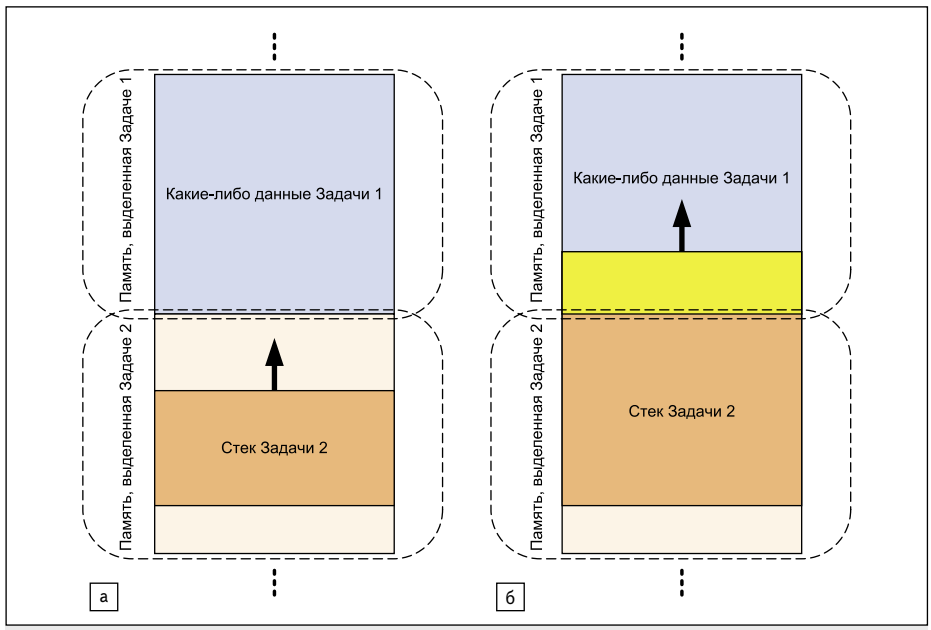

Рис. 8. Пример переполнения стека в многозадачной среде

Рассмотрим подробнее, в каких случаях расходуется память стека задачи:

1. При возникновении прерывания МК. Состояние процессора (то есть набор его регистров) и адрес возврата сохраняются в стеке задачи, и управление передается обработчику прерывания. Если архитектура МК поддерживает вложенные прерывания, то каждый раз при возникновении нового прерывания, когда обработчик «старого» еще не завершился, происходит еще одно сохранение в стеке текущей задачи состояния процессора.
2. При вызове функции. Помимо текущего состояния процессора и адреса возврата, в стек помещаются аргументы вызываемой функции. Вызываемая функция в свою очередь тоже может вызывать функции, что приводит к соответствующей записи в стек.
3. При использовании локальных переменных. Все локальные переменные функции (а следовательно, и задачи, так как задача представлена в программе в виде функции языка Си) по умолчанию располагаются в памяти стека. Обработчик прерывания в программе на Си также представлен функцией, поэтому переменные, объявленные внутри обработчика прерывания, также размещаются в стеке.

Переполнение стека — это ситуация, когда в результате рассмотренных выше случаев выделенного объема памяти для стека оказывается недостаточно, и данные, которые должны были попасть в стек задачи, записываются за пределами отведенной для этого области. Распределение памяти в случае переполнения стека показано на рис. 8б.

На рис. 8а в стек Задачи 2 были записаны какие-то данные, однако их объем оказался в пределах выделенной Задаче 2 области памяти для размещения стека. На рис. 8б в результате дальнейшего роста стека Задачи 2 размер стека превысил отведенный ему диапазон, из-за чего были повреждены данные, принадлежащие Задаче 1. Причем эффект от такого повреждения предсказать практически невозможно.

Таким образом, переполнение стека нарушает один из основных принципов работы многозадачной системы — принцип независимого выполнения нескольких задач на одном процессоре. Переполнение стека — одна из наиболее частых причин краха программы не только во встраиваемых приложениях.

Крайне сложно выявить возможность возникновения переполнения стека аналитически, поэтому на практике прибегают к экспериментальному определению размера стека, необходимого задаче, а также к механизмам обнаружения факта переполнения, что значительно облегчает поиск причины некорректной работы программы.

FreeRTOS предоставляет два метода обнаружения (и, возможно, коррекции последствий) переполнения стека. Для выбора одного из методов следует задать значение макроопределению ***configCHECK_FOR_STACK_OVERFLOW***, равное 1 или 2, в файле ***FreeRTOSConfig.h***. 0 означает отключенный механизм контроля переполнения стека.

Важно, что контроль переполнения стека возможен только на архитектурах, память которых не разбита на сегменты (например, порт для x86-процессоров в реальном режиме не допускает использования механизмов контроля переполнения стека, так как x86-процессоры имеют сегментированную модель памяти). Кроме того, некоторые процессоры могут генерировать аппаратное исключение в ответ на переполнение стека до того, как оно будет обработано средствами FreeRTOS.

Если включена опция контроля над переполнением стека, то в случае его возникновения автоматически вызывается функция обратного вызова ***vApplicationStackOverflowHook()***, которая должна содержать действия по индикации и, возможно, устранению последствий переполнения стека. Если включена опция контроля переполнения стека, то программа должна содержать определение функции ***vApplicationStackOverflowHook()***. Ее прототип:

```
void vApplicationStackOverflowHook( xTaskHandle *pxTask, signed portCHAR *pcTaskName );
```

В аргументы ***pxTask*** и ***pcTaskName*** автоматически помещаются, соответственно, указатель на дескриптор и имя задачи, стек которой переполнился. Однако следует помнить, что эффект от переполнения стека может быть разным, в том числе и таким, что аргументы функции ***vApplicationStackOverflowHook()*** могут оказаться искаженными. В этом случае надежнее прочитать значение глобальной переменной ***pxCurrentTCB***, которая определена в файле ***tasks.c*** и содержит указатель непосредственно на блок управления (на структуру ***tskTCB***) той задачи, которая в данный момент выполняется.

Помните, что задействованный контроль переполнения стека приводит к дополнительному расходу времени процессора каждый раз при переключении с одной задачи на другую, поэтому использовать его следует только на этапе разработки и тестирования программы.

Метод контроля переполнения стека № 1

Потребление памяти стека с большой долей вероятности достигает своего максимального значения в тот момент, когда задача выходит из состояния выполнения, так как стек задачи хранит ее контекст. В этот момент ядро может проверить регистр «Указатель стека» микроконтроллера на предмет, выходит ли его значение из заданного диапазона. Функция обратного вызова ***vApplicationStackOverflowHook()*** вызывается, если значение указателя стека превышает объем стека, выделенный задаче при ее создании.

Это не гарантирует отслеживание 100% всех случаев переполнения стека. Например, данный метод не выявит переполнение стека, вызванное одновременным срабатыванием большого числа прерываний.

Для активации этого метода значение макроопределения ***configCHECK_FOR_STACK_OVERFLOW*** следует задать равным 1.

Метод контроля переполнения стека № 2

Ядро выполняет контроль переполнения стека по методу № 1, но дополнительно выполняет следующую проверку.

Когда задача создается, ее стек побайтно заполняется известным значением (конкретно значением 0xA5). Когда задача покидает состояние выполнения, ядро проверяет верхние 16 байт памяти ее стека. Если хотя бы один байт из них оказался не равен изначально записанному известному значению (из-за активности задачи или прерывания), то делается вывод, что стек был переполнен, и вызывается функция ***vApplicationStackOverflowHook()***.

Данный метод более эффективен, чем первый, но требует и больше времени на выполнение. Его особенность в том, что, если верхушка стека достигнет области последних 16 байт выделенной для него памяти, но не превысит выделенного объема, то метод укажет на переполнение стека, которого в сущности не было.

Как и первый, этот метод не гарантирует фиксацию всех случаев переполнения стека. Для активации контроля по методу № 2 значение макроопределения ***nfigCHECK_FOR_STACK_OVERFLOW*** следует задать равным 2.

Пример контроля переполнения стека

Рассмотрим практическое применение контроля переполнения стека на примере контроллера бесколлекторного двигателя постоянного тока, описанного выше. Цель — выяснить, наступает ли в программе ситуация переполнения стека любой из задач, и, соответственно, подобрать такой объем выделяемого задаче стека, чтобы ситуация переполнения не наступала.

В программе создаются три задачи следующим образом:

```
/* Создание задач */
    xTaskCreate(vAccelTask,
                ( signed char * ) "Accel",
                configMINIMAL_STACK_SIZE + 100,
                NULL,
                PRIORITY_ACCEL_TASK,
                NULL );
    xTaskCreate(vServoTask,
                ( signed char * ) "Servo",
                configMINIMAL_STACK_SIZE + 100,
                NULL,
                PRIORITY_SERVO_TASK,
                NULL );
    xTaskCreate(vProcessingTask,
                ( signed char * ) "Proc",
                configMINIMAL_STACK_SIZE + 100,
                NULL,
                PRIORITY_PROCESSING_TASK,
                NULL );
```

Размер минимального достаточного размера стека ***configMINIMAL_STACK_SIZE*** взят равным 85 байт, так, как рекомендуется в демонстрационном проекте из дистрибутива FreeRTOS.

Включим контроль стека по методу № 2, добавив в файл ***FreeRTOSConfig.h*** строку:

```
#define configCHECK_FOR_STACK_OVERFLOW 2
```

Определим функцию обратного вызова, которая автоматически вызывается при переполнении стека:

```
void vApplicationStackOverflowHook(xTaskHandle *pxTask, signed
portCHAR *pcTaskName ) {
    /* Инвертировать вывод общего назначения PG3. */
    DDRG  |= (1 << DDG3);
    PORTG  ^= (1 << PG3);
}
```

Таким образом, переполнение стека любой задачи в системе будет приводить к инвертированию вывода МК, к которому подключен светодиод.

Если скомпилировать программу, загрузить ее в МК и подать на него питание, то по мигании светодиода можно видеть, что переполнение стека происходит периодически.

Зададим размер стека для каждой задачи на 100 байт больше, то есть равным ***configMINIMAL_STACK_SIZE + 200***. После подачи питания можно видеть, что проблема не исчезла: светодиод продолжает мигать.

Зададим значение макроопределения ***configMINIMAL_STACK_SIZE*** равным 90 байт вместо 85. Прогон программы показывает, что проблема переполнения стека исчезла.

Такое поведение объясняется тем, что в программе автоматически создается задача Бездействие, причем размер стека для нее задается равным ***configMINIMAL_STACK_SIZE***. Очевидно, что этого размера оказалось недостаточно, особенно в моменты возникновения прерываний микроконтроллера. Поэтому возникновение прерываний во время выполнения задачи Бездействие приводило к переполнению стека.

Таким образом, в результате использования механизма контроля переполнения стека удалось выяснить, что для процессора ATmega128L и конкретно данного приложения минимальный размер стека должен составлять 90 байт вместо 85, как указано в демонстрационном проекте.

Тем не менее если активировать контроль по методу № 1:

```
#define configCHECK_FOR_STACK_OVERFLOW 1
```

то прогон программы показывает отсутствие переполнения, даже при значении ***configMINIMAL_STACK_SIZE***, равном 85. Это может говорить как о том, что при контроле по методу № 2 переполнения стека не было, а была лишь запись данных в последние 16 байт области стека, так и о том, что метод № 1 не выявил факт переполнения.

Необходимо оставить некоторый «запас прочности» по минимальному объему стека, поэтому в рабочей программе он равен 90 байт.

Отладка программы с помощью StateViewer

Компания WITTENSTEIN, занимающая продвижением коммерческих версий FreeRTOS — OpenRTOS и SafeRTOS, предлагает утилиту StateViewer, которая встраивается в среду разработки Eclipse, а также в среды разработки фирм IAR и Keil.

Утилита позволяет получить в режиме реального времени отладочную информацию обо всех задачах и всех очередях (и объектах, на них основывающихся, — семафорах и мьютексах) при отладке встраиваемого приложения в вышеперечисленных средах разработки. Кроме OpenRTOS и SafeRTOS, эта утилита работает и с FreeRTOS.

Утилита StateViewer бесплатна, ее можно загрузить с [4], указав свой ящик электронной почты.

Установка утилиты подробно описана в документации на нее, которая поставляется вместе с самой утилитой, поэтому останавливаться на этом не будем.

Рассмотрим пример работы с утилитой StateViewer на примере среды разработки Eclipse. Для того чтобы получить отладочную информацию, достаточно перейти в режим отладки приложения и вызвать на экран два окна: ***Task Table*** (рис. 9) и ***Queue Table*** (рис. 10), представляющие информацию, соответственно, о задачах и очередях в программе. Сделать это можно выбрав пункт меню ***Window → Show View → Other… → OpenRTOS Viewer***. Далее утилита сама соберет информацию о существующих в программе задачах и очередях. Информация в окнах ***Task Table*** и ***Queue Table***

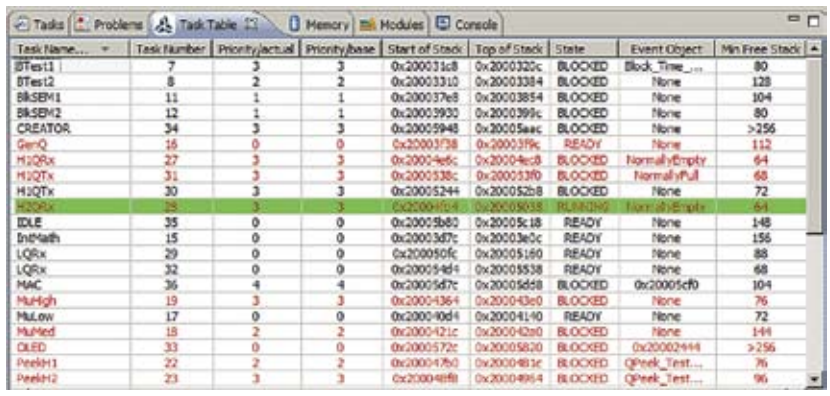

Рис. 9. Окно Task Table утилиты StateViewer

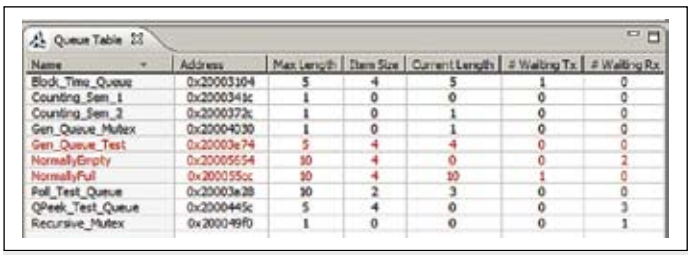

Рис. 10. Окно Queue Table утилиты StateViewer

обновляется только после приостановки выполнения программы (например, с помощью точки останова).

Что касается самой FreeRTOS, то для того, чтобы очереди были представлены именами (как показано на рис. 10), необходимо связать очередь, заданную своим дескриптором, с ее именем в виде текстовой строки. В терминах FreeRTOS это называется добавлением очереди в реестр. Для этого предназначена API-функция ***vQueueAddToRegistry()***. Ее прототип:

```
void vQueueAddToRegistry(xQueueHandle xQueue,
                         signed portCHAR *pcQueueName );
```

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

1. ***xQueue*** — дескриптор очереди (семафора, мьютекса), которую необходимо добавить в реестр.
2. ***pcQueueName*** — имя очереди (семафора, мьютекса), заданное в виде нуль-терминальной строки. Именно эта строка будет отображаться в списке очередей в окне ***Queue Table***.

Стоит отметить, что API-функция ***vQueueAddToRegistry()*** не имеет другого предназначения, кроме как для целей отладки. Причем добавлять в реестр следует только те очереди, поведение которых необходимо выяснить в ходе отладки. Макроопределение ***configQUEUE_REGISTRY_SIZE*** в файле ***FreeRTOSConfig.h*** задает максимальное количество очередей, которые могут быть добавлены в реестр.

Пример добавления очереди в реестр:

```
void vAFunction( void )
{
xQueueHandle xCharQueue;

    /* Создать очередь для хранения 10 символов. */
    xCharQueue = xQueueCreate( 10, sizeof( portCHAR ) );

    /* Занести ее в реестр, таким образом мы сможем наблюдать ее в StateViewer.
    под именем, соответствующим имени очереди в программе. */
    vQueueAddToRegistry(xCharQueue, "xCharQueue" );
}
```

Если очередь добавлена в реестр и программа предусматривает ее удаление, то прежде чем удалять, очередь необходимо исключить из реестра. Для этого предназначена API-функция ***vQueueUnregisterQueue()***. Ее прототип:

```
void vQueueUnregisterQueue( xQueueHandle xQueue );
```

Единственный аргумент ***xQueue*** — дескриптор очереди, которую необходимо исключить из реестра. Таким образом, процедура удаления очереди, которая ранее была помещена в реестр, выглядит так:

```
vQueueUnregisterQueue( xQueue );
vQueueDelete( xQueue );
```

Список задач Task Table

Представляет собой список всех задач, созданных на данный момент в программе. Каждой задаче соответствует одна строка таблицы (рис. 9), причем зеленым цветом выделена выполняющаяся в данный момент задача. Красным цветом помечены те задачи, которые изменили свое состояние с момента последней приостановки выполнения программы. Каждому параметру задачи соответствует один столбец таблицы. Список параметров:

1. ***Task Name*** — имя задачи, назначенное ей в момент создания (аргумент API-функции ***xTaskCreate()***).
2. ***Task Number*** — уникальный номер задачи, который автоматически был назначен ей ядром.
3. ***Priority/actual*** — приоритет задачи в данный момент. Работа механизма наследования приоритетов может стать причиной временного повышения приоритета задачи по сравнению с приоритетом, с каким задача была создана.
4. ***Priority/base*** — приоритет задачи, назначенный ей в момент создания или измененный с помощью вызова API-функции ***vTaskPrioritySet()***.
5. ***Start of Stack*** — адрес начала стека задачи. Стек заполняется начиная именно с адреса начала стека.
6. ***Top of Stack*** — адрес вершины стека задачи в данный момент. Именно по этому адресу был сохранен контекст задачи, находящейся в блокированном или приостановленном состоянии.
7. ***State*** — текущее состояние задачи. Может принимать следующие значения:
   - ***Running*** — выполняется в данный момент.
   - ***Ready*** — готова к выполнению.
   - ***Blocked*** — блокирована.
   - ***Suspended*** — приостановлена.
8. ***Event Object*** — имя или адрес очереди (семафора, мьютекса), ожидание операции с которой послужило причиной перехода задачи в блокированное состояние.
9. ***Min Free Stack*** — минимальный объем свободной памяти стека за все время отладки программы. Равен минимальной разнице между адресом конца стека, который определен в момент создания задачи, и адресом вершины стека, который «плавает» в зависимости от вызова функций, срабатывания прерываний и сохранения контекста задачи.

Вычисление объема свободного стека занимает существенное время, поэтому можно отключить эту возможность, щелкнув правой кнопкой в любом месте окна ***Task Table*** и выбрав ***Toggle Stack Checking***.

Список очередей Queue Table

Как и в случае списка задач, красный цвет обозначает произошедшие изменения в состоянии очереди с момента последней остановки программы (рис. 10). Для каждой очереди выводятся следующие поля:

1. ***Name*** — имя очереди, которое было присвоено ей в момент, когда она была помещена в реестр.
2. ***Address*** — адрес структуры управления очередью, он же — дескриптор очереди.
3. ***Max Length*** — размер очереди, то есть максимальное количество элементов, которые одновременно может хранить очередь. Определяется в момент создания очереди.
4. ***Item Size*** — размер одного элемента очереди в байтах.
5. ***Current Length*** — длина очереди, то есть количество элементов, которые в данный момент хранятся в очереди. Всегда меньше размера очереди.
6. ***# Waiting Tx*** — количество задач, которые заблокировались, ожидая, когда в очереди появится свободное место (когда появится возможность записи в очередь).
7. ***# Waiting Rx*** — количество задач, которые заблокировались, ожидая, когда в очередь будет записан хотя бы один элемент (когда появится возможность прочитать элемент из очереди).

Выводы

Отладка многозадачной программы в условиях небольшого объема оперативной памяти (что свойственно микроконтроллерам) представляет собой трудную задачу. Тем не менее FreeRTOS предлагает богатый инструментарий, позволяющий существенно ее об-

легчить. FreeRTOS предлагает встроенные механизмы:

- трассировка задач и событий ядра;
- сбор статистики выполнения задач;
- отслеживание объема стека, потребляемого задачей.

Автор выражает огромную благодарность Евгению Онищуку за помощь в сборке и наладке опытного образца контроллера бесколлекторного двигателя.

Литература

1. Курниц А. FreeRTOS — операционная система для микроконтроллеров // Компоненты и технологии. 2011. № 2–10.
2. http://www.freertos.org
3. http://ru.wikipedia.org/wiki/Порядок_байтов
4. http://www.highintegritysystems.com/index.php? option=com_chronocontact&Itemid=67