Василий @qdx
Пользователь

26 сентября 2011 в 17:30

FreeRTOS: межпроцессное взаимодействие

Программирование микроконтроллеров*

Здравствуйте. В данной статье я постараюсь описать метод межпроцессного обмена данными и синхронизацию с эвентами.
Ссылки на остальные части:
FreeRTOS: введение.
FreeRTOS: мьютексы и критические секции.

Любая многопоточная ОС, не будет считаться полной, без соответствующих средств поддержки многопоточного окружения. FreeRTOS облада всем необходимым для этого, а именно:

- Очереди, для обмена данными между тасками, или ISR.
- Бинарные семафоры, и счетные семафоры для синхронизации с эвентами (прерываниями).
- Мьютексы, для совместного доступа к ресурсу (например, порт).
- Критические секции, для создания области кода, выполнение которой не может быть прервано планировщиком.

Очереди.

Программы, написанные с помощью FreeRTOS представляют собой набор независимых тасков, или миниподпрограмм, которым требуется эффективный и потокобезопасный механизм обмена данными, в случае FreeRTOS — это очереди.
Очередь — это простой FIFO(хотя также можно писать в начало очереди, а не в конец) буфер, который может хранить фиксированное число элементов, известного размера. Запись в очередь — это побайтовое копирование данных в буфер, чтение — копирование данных и удалени очереди.
Очереди — это, по сути, независимые объекты, которые могут иметь множество писателей, и читателей, без боязни прочитать\записать бить данные. При чтении данных, опционально, мы можем указать время, в течение которого таск должен находится в ожидании получения новы данных. При записи данных мы также можем указать данное время, но уже для ожидания места в очереди.

Рассмотрим более подробно основные функции по работе с очередями в FreeRTOS.
Перед использованием любой очереди, она должна быть создана. RAM память для очереди выделяется из FreeRTOS хипа, и ее размер равен размер данных+размер структуры очереди. В коде каждая очередь представлена ее хэндлом, типа xQueueHandle.

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

uxQueueLength — максимальное число элементов, которое очередь может хранить в единицу времени.
uxItemSize — размер каждого элемента очереди.
return xQueueHandle, или NULL — в случае если очередь создана будет возвращен соответствующий хэндл, если нет, т.е. недостаточно п то будет возвращен NULL.

Для записи в очередь используют, также специальные функции:

```
portBASE_TYPE xQueueSendToFront(xQueueHandle xQueue, const void * pvItemToQueue, portTickType xTicksToWait);
portBASE_TYPE xQueueSendToBack(xQueueHandle xQueue, const void * pvItemToQueue, portTickType xTicksToWait); // Или эквивалент portBASE_TYPE xQu
d(xQueueHandle xQueue, const void * pvItemToQueue, portTickType xTicksToWait);
```

xQueue — хэндл очереди, в которую записываем данные.
pvItemToQueue — указатель на элемент, который будет помещен в очередь.
xTicksToWait — время, в течение которого таск должен находиться в заблокированном состоянии, чтобы появилось место в очереди. Можн указать **portMAX_DELAY**, чтобы таск находился в блокированном состоянии в течение неопределенного времени, т.е. пока не появится мес очереди.
return pdPASS, или errQUEUE_FULL — в случае, если новый элемент успешно записан в очередь, то функция возвращает pdPASS, если м достаточно, и указано время xTicksToWait, то таск перейдет в заблокированное время, для ожидания места в очереди.

Для чтения данных, используются 2 функции, основное отличие, которых в том, что xQueueReceive удаляет элемент из очереди, а xQueuePe нет.

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

xQueue — хэндл очереди, в которую записываем данные.
pvBuffer — указатель на буфер памяти, в который будут прочитаны данные из очереди. Тип буфера=типу элементов очереди.
xTicksToWait — время, в течение которого таск должен находиться в заблокированном состоянии, чтобы данные появились в очереди. Мож

указать **portMAX_DELAY**, чтобы таск находился в блокированном состоянии в течение неопределенного времени, т.е. пока не появятся нов данные в очереди. Далее я расскажу, как это используют для создания Gatekeeper Task.
return pdPASS, или errQUEUE_EMPTY — в случае, если новый элемент успешно прочитан из очереди, то функция возвращает pdPASS, ес очередь пуста, и указано время xTicksToWait, то таск перейдет в заблокированное время, для ожидания новых данных в очереди.

Для просмотра количества элементов в очереди, можно использовать функцию

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

Важно: вышерассмотренные функции, нельзя использовать в ISR(прерываниях) и для них существуют, специальные версии со специальным суффиксом **ISR**, поведение которых аналогично предыдущим функциям, за исключением последнего параметра:

```
portBASE_TYPE xQueueSendToFrontFromISR( xQueueHandle xQueue, void *pvItemToQueue, portBASE_TYPE *pxHigherPriorityTaskWoken );
portBASE_TYPE xQueueSendToBackFromISR(xQueueHandle xQueue, void *pvItemToQueue, portBASE_TYPE *pxHigherPriorityTaskWoken);
portBASE_TYPE xQueueReceiveFromISR(xQueueHandle xQueue, const void *pvBuffer, portBASE_TYPE *pxHigherPriorityTaskWoken);
```

pxHigherPriorityTaskWoken — так как запись в очередь может привести к разблокированию таска, ожидающего данных и имеющего больш приоритет, чем текущий таск, то нам необходимо выполнить форсированное переключение контекста (для этого необходимо вызвать макрос taskYIELD()). В случае необходимости данный параметр будет равен **pdTRUE**.

Пару слов об эффективном использовании очередей. Например, рассмотрим UART — при типичном подходе, каждый полученный байт сразу записывают в очередь, что делать не стоит т.к. это жутко неэффективно, уже на достаточно небольших частотах. Более эффективно — пров базовую обработку данных в ISR и затем передавать их в очередь, но важно понимать — код ISR при этом должен быть как можно короче.

Рассмотрим пример, так называемого gatekeeper task(не знаю, как корректно перевести это, если кто подскажет, буду благодарен:)).
Gatekeeper task — это простой метод, который позволяет избежать главных проблем многопоточного программирования: инвертирования приоритетов, и попадания таска в тупик(deadlock).
Gatekeeper task — это единственный метод, который имеет прямой доступ к ресурсу, все остальные таски должны обращаться к ресурсу чер таск.
Рассмотрим, простой скелет gatekeeper task. Это чисто надуманный пример, но который поможет понять общий принцип. Примем, что нам необходимо безопасно отправлять некоторые данные, например, через UART.

```
// Gatekeeper фактически является обычным таском.
void vGatekeeperTask( void *pvParameteres ) {
    char oneByte;
    // Данный таск - это единственное место, которое будет иметь доступ к UART порту, и все остальные таски должны передавать данные через него
    for( ;; ) {
                // Но его тело отличается.
                // Так как мы указали portMAX_DELAY в качестве времени ожидания, то нет необходимости проверять возвращаемое значение.
                // Возврат из функции произойдет только, если данные прочитаны из очереди.
            xQueueReceive( xDataQueue, &oneByte, portMAX_DELAY);
                // Нижеследующий код исполниться только, если мы получили новые данные.
                vSendByteToUART( oneByte );
                // Переходим в режим ожидания.
        }
}
```

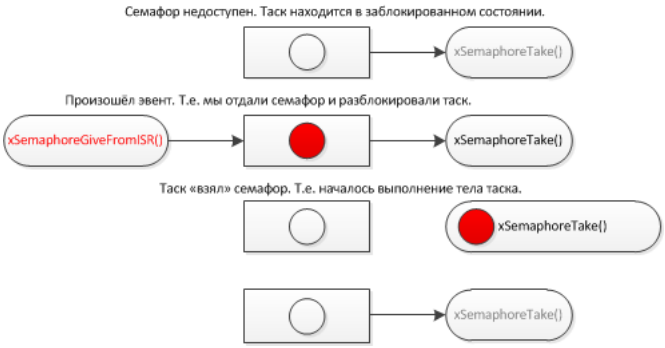
Таким образом, любой таск, который захочет отправить байт с помощью UART может использовать, одну из служебных функций, например,

```
void vUARTPutByte( char byte) {
    // Я указал время ожидание равное нулю, хотя можно было также установить некоторое значение, на случай переполнения очереди.
    xQueueSend( xDataQueue, &byte, 0 );
}
```

Стоит также отметить, что не стоит ограничиваться, только char в качестве типа данных очереди, а можно организовывать целые конвейеры используя структуры.

Бинарные семафоры.

Бинарные семафоры, могут использоваться для разблокирования таска, всякий раз, когда происходит какой-либо эвент (например, нажатие кнопки).
Типичный сценарий работы: при попадании в ISR определенного прерывания, мы *отдаем* семафор, в результате таск, ожидающий семафора *забирает* семафор и выходит из разблокированного состояния для проведения каких-либо операций. Данный механизм, показан на следующ рисунке.



Для хранения всех типов семафоров, используется тип данных *xSemaphoreHandle*.
Рассмотрим функции для работы с семафорами:

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

xSemaphore — данная функция реализована с помощью макроса, поэтому необходимо передавать значение хэндла, а не указатель на него случае успешного создания семафора, значение xSemaphore не равно NULL.

Для того чтобы «взять» семафор используется специальная функция:

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

xSemaphore — хэндл семафора, который планируется «взять».
xTicksToWait — время, в течение которого таск должен находиться в заблокированном состоянии, по истечении которого семафор станет доступен. Можно указать **portMAX_DELAY**, чтобы таск находился в заблокированном состоянии в течение неопределенного времени, т.е. по семафор не будет доступен.
return pdPASS, или pdFALSE — pdPASS — если семафор получен, pdFALSE — если семафор недоступен.

Для того чтобы «отдать» семафор, также используются специальные функции. Я рассмотрю ISR версию т.к. наиболее часто семафоры приме связке с ISR.

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

xSemaphore — хэндл семафора, который планируется «отдать».
pxHigherPriorityTaskWoken — так как «передача» семафора может привести к разблокированию таска, ожидающего данных семафора им большой приоритет, чем текущий таск, то нам необходимо выполнить форсированное переключение контекста (для этого необходимо вызва макрос taskYIELD()). В случае необходимости данный параметр будет равен **pdTRUE**.
return pdPASS, или pdFALSE — pdPASS — если семафор отдан, pdFALSE — если семафор уже доступен, но не обработан.

В качестве примера, приведу короткий код программы, для ISR я написал псевдокод:

```
xSemaphoreHandle xButtonSemaphore;
void vButtonHandlerTask( void *pvParameteres ) {
    for( ;; ) {
            xSemaphoreTake( xButtonSemaphore, portMAX_DELAY );
                // Здесь нужно разместить код по нажатию кнопки.
        }
}
void main() {
    // Инициализация микроконтроллера
        vInitSystem();

        vSemaphoreCreateBinary( xButtonSemaphore );
    if( xButtonSemaphore != NULL ) {
            // Создание тасков. Я не включил код проверки ошибок, не стоит забывать об этом!
            xTaskCreate( &vButtonHandlerTask, (signed char *)"GreenBlink", configMINIMAL_STACK_SIZE, NULL, 1, NULL );

            // Запуск планировщика, т.е. начало работы тасков.
            vTaskStartScheduler();
        }

        // Сюда стоит поместить код обработки ошибок, в случае если планировщик не заработал, или семафор не был создан. Для простоты я использ
сто бесконечный цикл.
        for( ;; ) { }
}
ISR_FUNCTION processButton() {
   portBASE_TYPE xTaskWoken;
   if( buttonOnPressed ) {
       xSemaphoreGiveFromISR( xButtonSemaphore, &xTaskWoken );
```

```
            if( xTaskWoken == pdTRUE) {
                taskYIELD();
            }
        }
    }
```

Счётные семафоры.

Рассмотрим типичную ситуацию, которая существует при использовании семафоров:
1. Произошел какой-то эвент, который вызвал прерывание.
2. ISR «отдает» семафор, т.е. разблокирует ожидающий семафор таск.
3. Ожидающий таск «забирает» семафор.
4. После исполнения нужного кода таск опять переходит в блокированное состояние, ожидая новых эвентов.

Данный алгоритм отлично работает, но только не на больших частотах. На больших частотах необходимо использовать счетные семафоры, которые, как правило, используют в 2-х случаях:

- Также для синхронизации с эвентами, но на больших частотах.
- Управление ресурсами. В данном случае, количество семафоров обозначает количество доступных ресурсов.

Как указывалось выше, для хранения всех типов семафоров используется тип данных xSemaphoreHandle, и так как перед использованием семафора, он должен быть создан, то необходимо использовать специальную функцию для создания счетных семафоров:

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

uxMaxCount — максимальное количество семафоров, которое может хранить счетчик. По аналогии с очередью — это длина очереди.
uxInitialCount — значение счетчика после создания семафора.
return не NULL — функция возвращает не NULL значение, если семафор был создан.
В остальном для работы со счетными семафорами используются функции, аналогичные предыдущим.