

25 сентября 2011 в 01:08

## FreeRTOS: введение



Здравствуйте. В короткой серии постов я постараюсь описать возможности, и подходы работы с одной из наиболее популярной и развивающейся РТОС для микроконтроллеров – FreeRTOS. Я предпологаю базовое знакомство читателя с теорией многозадачности, о которс можно почитать в одном из соседних постов на Хабре или ещё где-то.

Ссылки на остальные части:

FreeRTOS: межпроцессное взаимодействие. FreeRTOS: мьютексы и критические секции.

Зачем все это? Или введение в многозадачные системы, от создателей FreeRTOS.

Традиционно существует 2 версии многозадачности:

- «Мягкого» реального времени(soft real time)
- «Жесткого» реального времени(hard real time)

К ОСРВ мягкого типа можно отнести наши с Вами компьютеры т.е. пользователь должен видеть, что, например, нажав кнопку с символом, он введенный символ, а если же он нажал кнопку, и спустя время не увидел реакции, то ОС будет считать задачу «не отвечающей» (по аналоги Windows — «Программа не отвечает»), но ОС остается пригодной для использования. Таким образом, ОСРВ мягкого времени просто определ предполагаемое время ответа, и если оно истекло, то ОС относит таск к не отвечающим.

К ОСРВ жесткого типа, как раз относят ОСРВ во встраиваемых устройствах. В чем-то они похоже на ОСРВ на дестопах(многопоточное выполна одном процессоре), но и имеют главное отличие — каждая задача **должна** выполняться за отведенный квант времени, не выполнение да условия ведет к краху все системы.

А все таки зачем?

Если у Вас есть устройство с нетривиальной логикой синхронизации обмена данными между набором сенсоров, если Вам действительно нуж гарантировать время отклика, и наконец-то если Вы думаете, что система может разростись, но не знаете насколько, то РТОС Ваш выбор.

Не стоит применять РТОС, для применения РТОС т.е. не нужно применять РТОС в слишком тривиальных задачах(получить данные с 1 сенсор отправить дальше, обработать нажатие 1 кнопки и т.д) т.к. это приведет к ненужной избыточности, как полученного кода, так и решения са задачи.

Работа с тасками(или задачами, процессами).

Для начала приведу несколько определений, для того чтобы внести ясность в дальнейшие рассуждения:

"Операционные системы реального времени (OCPB(RTOS)) предназначены для обеспечения интерфейса к ресурсам критических по вресистем реального времени. Основной задачей в таких системах является своевременность (timeliness) выполнения обработки данных".

"FreeRTOS — многозадачная операционная система реального времени (OCPB) для встраиваемых систем. Портирована на несколько микропроцессорных архитектур.

От хабраюзера @andrewsh, по поводу лицензии: разрешено не публиковать текст приложения, которое использует FreeRTOS, несмотря на то ОS линкуется с ним. Исходники самой же RTOS должны всегда прикладываться, изменения, внесённые в неё — тоже.".

FreeRTOS написана на Си с небольшим количеством ассемблерного кода (логика переключения контекста) и ее ядро представлено всего 3-м: файлами. Более подробно о поддерживаемых платформах можно почитать на официальном сайте.

Перейдем к делу.

Любой таск представляет собой Си функцию со следующим прототипом:

```
void vTask( void *pvParametres );
```

Каждый таск – это по сути мини подпрограмма, которая имеет свою точку входу, и исполняется внутри бесконечного цикла и обычно не дол: выходить из него, а также имеет собственный стэк. Одно определение таска может использоваться для создания нескольких тасков, которык выполняться независимо и также иметь собственный стэк.

Тело таска не должно содержать явных **return;** конструкций, и в случае если таск больше не нужен, его можно удалить с помощью вызова А функции. Следующий листинг, демонстрирует типичный скелет таска:

```
void vTask ( void *pvParametres) {

/* Данный фрагмент кода будет вызван один раз, перед запуском таска.

Каждый созданный таск будет иметь свою копию someVar.

Кроме объявления переменных, сюда можно поместить некоторый инициализационный код.

*/

int someVar;

// Так как каждый таск - это по сути бесконечный цикл, то именно здесь начинается тело таска.

for(;;) {

// Тело таска
}

// Так как при нормальном поведении мы не должны выходить из тела таска, то в случае если это все таки произошло, мы удаляем таск.

// Функция vTaskDelete принимает в качестве аргумента хэндл таска, который стоит удалить.

// Вызов внутри тела таска с параметром NULL,удаляет текущий таск

vTaskDelete( NULL );
```

Для создания таска, и добавления ее в планировщик используется специальная АРІ функция со следующим прототипом:

*pvTaskCode* – так как таск – это просто Си функция, то первым параметром идет ее значение.

pcName - имя таска. По сути это нигде не используется, и полезно только при отладке с соответствующими плагинами для IDE.

**usStackDepth** – так как каждый таск – это мини подпрограмма, со своим стэком, то данный параметр отвечает за его глубину. При скачиван RTOS и разворачивания системы для своей платформы, вы получаете файл FreeRTOSConfig.h настройкой которого можно конфигурировать поведение самой ОС. В данном файле также объявлена константная величина **configMINIMAL\_STACK\_SIZE**, которую и стоит передавать в качестве usStackDepth с соответствующим множителем, если это необходимо.

**pvParameters** – при создании, каждый таск может принимать некоторые параметры, значения, или ещё что-то что может понадобиться вну тела самого таска. С точки зрения инкапсуляции, этот подход наиболее безопасный, и в качестве pvParameters стоит передавать, например, некоторую структуру, или NULL, если ничего передавать не нужно.

*uxPriority* – каждый таск имеет свой собственный приоритет, от 0(min) до (**configMAX\_PRIORITIES** – 1). Так как, по сути, нет верхнего пр для данного значения, то рекомендуется использовать как можно меньше значений, чтобы не было дополнительно расхода RAM на данную *у* 

**pxCreatedTask** — хэндл созданного таска. При создании таска, опционально можно передать указатель на хэндл будующего таска, для последующего управления работой самого таска. Например, для удаления определенного таска.

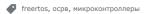
Данная функция возвращает **pdTRUE**, в случае успешного создания таска, или **errCOULD\_NOT\_ALLOCATE\_REQUIRED\_MEMORY**, в случає размер стэка был указан слишком большим, т.е. недостаточно размера хипа для хранения стэка таска, и самого таска.

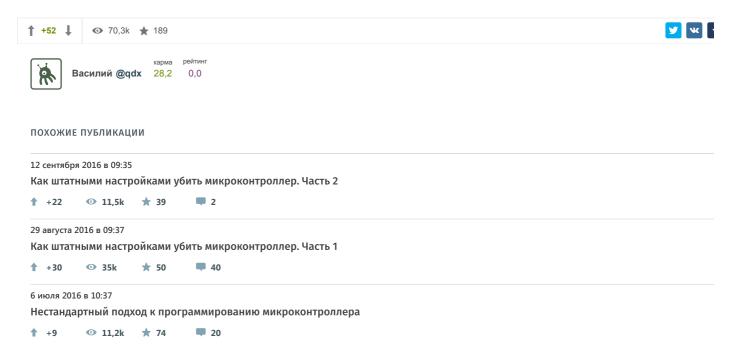
На следующем листинге я привел, короткий пример законченной программы, которая создает 2 таска, каждый из которых мигает светодиодо

```
25.07.2017
```

```
void main(void) {
       // Инициализация микроконтроллера. Данный код будет у каждого свой.
       // Создание тасков. Я не включил код проверки ошибок, но не стоит забывать об этом!
       xTaskCreate( &vGreenBlinkTask,
                    (signed char *) "GreenBlink",
                    configMINIMAL_STACK_SIZE,
                    NULL,
                    1,
                    NULL );
       xTaskCreate( &vRedBlinkTask,
                    (signed char *) "RedBlink",
                    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.
       // Для примера я использую просто бесконечный цикл.
       for(;;) { }
}
```

В следующем посте я планирую написать о взаимодействии между тасками, и работе с прерываниями.





CAMOE ЧИТАЕМОЕ Paspa

