BuguRTOS

noobs guide

anonimous

Здравствуйте %username%

Если Вы читаете этот поток сознания, значит Вас заинтересовала BuguRTOS. В этом документе Вы познакомитесь лишь с основными понятиями и получите инструкции по первым шагам в создании приложений на базе BuguRTOS. Документация по API BuguRTOS есть по адресу: https://drive.google.com/folderview?id=0B32mjehjqcIOYlFtNnRSc0JxdGc&usp=sharing

1 Что такое BuguRTOS?

BuguRTOS это ядро операционная система, которую пишет аноним в свободное время для УДОВОЛЬСТВИЯ.

Вы наверняка уже сталкивались с операционными системами на своем уютненьком компьютере, и думаете, что там обязательно есть куча программ, графическая оболочка или хотя бы консоль... Так вот, у меня для Вас плохие новости: разные люди понимают под Операционной Системой (ОС) разные вещи.

Кто-то называет ОС картинку на своем мониторе; кто-то — «Ядро» и минимальный набор программ и прикладных библиотек, реализующих функционал этого ядра; а кто-то — библиотеку, позволяющую запускать «много» процессов (потоков, задач и т.п.) на «небольшом» числе процессоров.

BuguRTOS в своем нынешнем состоянии — это именно такая библиотека! Фактически BuguRTOS — это набор исходных текстов, написанных на языках С и ассемблер, которые Вы можете включить в свой проект прошивки для Огромного Боевого Человекоподобного Робота.

2 Зачем мне это нужно?

Если Вы просто хотите помигать светодиодами своего Arduino™, то Вам это не нужно. Но если Вы собираетесь написать большой сложный проект на несколько тысяч строк кода и Вам требуется решать параллельно несколько задач, то тогда Вам вероятно пригодится ОС.

3 Но почему?

Представьте себе, что Вы пишете сложную программу для микроконтроллера, где надо управлять одновременно несколькими процессами.

И вот Вы натыкаетесь на процесс, не очень приоритетный, но требующий большого объема вычислений. Вы разработали элегантный алгоритм работы своего процесса, сделали его прямую реализацию, а в результате запороли работу остальных процессов... У Вас не хватило процессорного времени.

Что делать?

Правильно — надо сделать НЕ прямую реализацию алгоритма. И тут Вы начинаете РУКАМИ планировать работу процессора: разбиваете свой код на

маленькие куски и т.д.

А что в результате?

Правильно — сложная, ОЧЕНЬ СЛОЖНАЯ программа, которую трудно отлаживать и сопровождать. А представьте себе, что будет, если Вашу программу будет сопровождать кто-то другой?

И вот тут на помощь приходит операционная система. ОС позволяет вернутся к прямой реализации алгоритмов и тем самым снизить сложность программы. Естественно на работу ОС требуются ресурсы, но, если Вы научитесь правильно использовать средства ОС, то накладные расходы будут минимальны, а может еще и удастся освободить ресурсы процессора для решения еще большего количества задач.

4 Почему именно BuguRTOS?

Не почему, не нравится — есть возможность воспользоваться чем-то еще, сегодня написано много, ОЧЕНЬ МНОГО операционных систем, которые Вы можешь использовать в своих проектах (см. тут: http://en.wikipedia.org/wiki/List_of_real-time_operating_systems).

5 Лицензия?

GNU/GPL v 3, WITH LINKING EXCEPTION!

Да, друг мой, BuguRTOS можно использовать в закрытых проектах!

6 Но зачем GPL? Я же не смогу поменять код и закрыть его!

anonimous 23:14 07.01.2011

См. п. 4 и название ОС.

7 Какие архитектуры процессоров оно поддерживает?

В данный момент поддерживаются два семейства микроконтроллеров:

- Atmell AVR (GNU-toolchain);

- ST Microelectronics STM8 (IAR, Raisonance);
- Cortex-M0 (GNU-toolchain);
- Cortex-M3 (GNU-toolchain);
- Cortex-M4F (GNU-toolchain).

В планах поддержка других архитектур.

8 Какая текущая версия?

BuguRTOS-0.7.x: текущая версия 0.7.0. Задумывалась для применения в как в однопроцессорных, так и в многопроцессорных системах, но ввиду отсутствия порта на последние проверить это можно только при эмуляции многопроцессорности. Поддержка реальных многопроцессорных систем — вопрос времени.

9 Что оно может?

1. Многозадачность;
2. управление процессами (запустить, остановить, перезапустить сначала, продолжить исполнение с точки останова);
3. синхронизация процессов (сигналы, семафоры, мьютексы, IPC).

10 Что нового по отношению к прошлым версиям?

1. Система портирована на Cortex-M0,M3,M4F;
2. Удален протокол повышения приоритетов (HLP) в мьютексах (см. http://en.wikipedia.org/wiki/Priority_ceiling).
3. На замену HLP реализован протокол наследования приоритетов (см. http://en.wikipedia.org/wiki/Priority_inheritance);
4. Добавлена функция proc_set_prio, управляющая приоритетом процесса после его запуска.
5. Нарушена обратная совместимость API! Опять...

11 Средства разработки?

Для AVR как минимум потребуются:

1. avr-gcc;
2. binutils;
3. make;
4. avrdude;
5. программатор;

Опционально — IDE, я использую Code::Blocks.

Для STM8 потребуется IAR embedded workbench for STM8 версии 1.2 и выше, либо Raisonance tools for STM8.

Для Cortex-M потребуется GNU-toolchain (рекомендую: https://launchpad.net/gcc-arm-embedded), опционально IDE (Code::Blocks или Eclipse).

12 *Хорошо, я согласен «вернуть код с процентами». Как мне написать свой проект?*

1. Скачать исходные тексты из репозитория: svn checkout *http*://bugurtos.googlecode.com/svn/trunk/ bugurtos-read-only
2. Создать каталог в котором будут (я надеюсь) проекты на базе BuguRTOS (рабочее пространство).
3. Скопировать в рабочее пространство каталог bugurtos.
4. Создать в рабочем пространстве каталог для будущего проекта.
5. Создать проект.
6. Добавить файлы ОС в проект.
7. Написать файл config.h и добавить его в проект (для примера можно взять config.h из каталога bugurtos/arch/[archname]/).
8. Опции компиляции (Для AVR в конфигурации debug):

   -g
   -O0

   -mmcu=[%processor%]

   -std=c99

   -include "${PROJECT_DIR}/config.h"

   -I "[%BUGURTOS_DIR%]/arch/avr/gcc"

   -ffunction-sections

   -fdata-sections
9. Опции линковки (Для AVR в конфигурации debug):

   -mmcu=[%processor%]

   -Wl,-Map=$(TARGET_OUTPUT_FILE).map,--cref,--gc-sections
10. Написать main и все остальное, в качестве примера можно использовать main.c из тестов. Как минимум main.c должен содержать следующее:

```
// #include<bugurt.h> // Для Cortex-M
#include<bugurt_kernel.h> // Для AVR, STM8
/*Инклуды, дефайны, функции и переменные Вашего проекта*/

void idle_main(void * arg ){
	/*Опционально можно написать свою функцию idle_main*/
	while(1);
}
int main(void){
	init_bugurt();
	/*Инициализация периферии, системы прерываний, процессов*/
	start_bugurt();
	/*
	Выхода из start_bugurt() не произойдет, дальше можно ничего не писать, все равно исполняться не будет!!!
	*/
}
```

11.??????

12.PROFIT!!!

Внутри BuguRTOS

Многозадачная операционная система, это прежде всего планировщик и базовые инструменты управления процессами (задачами, потоками и т.п.), обо всем этом далее.

1 Процесс

В разных ОС это называется по разному: процесс, поток, задача и пр., суть такова: это независимый поток исполнения инструкций процессора.

То есть это исполняющийся кусок Вашей программы, у которого есть своя собственная главная функция «pmain», и эта «pmain» может быть написана так, как будто других процессов нет!

Каждый процесс имеет свой собственный стек, что такое стек и зачем он нужен — смотрите в википедии.

Процессы в BuguRTOS можно разделить на процессы «реального времени» и процессы «общего назначения», отличаются они алгоритмом планирования

исполнения, см. планировщик.

Процессы могут иметь общие ресурсы. Например, 2 или более запущенных процессов могут использовать 1 и ту же функцию «pmain» в этом случае можно говорить о нескольких запущенных экземплярах «pmain».

Когда планировщик прекращает выполнение процесса он вызывает функцию sv_hook, а когда возобновляет — rs_hook.

Как аргумент функциям pmain, rs_hook, sv_hook передается указатель типа void *. Для хранения локальных данных каждого экземпляра pmain можно использовать специализированную структуру или просто массив, указатель на такое хранилище данных можно передать в качестве аргумента в эти функции.

Процессы могут иметь общие данные, в этом случае доступ к таким данным необходимо осуществлять с помощью средств синхронизации, предоставляемых ОС, в противном случае такие данные могут быть повреждены при т.н. конкурентном доступе: это когда 2 процесса пытаются одновременно записать данные, или когда 1 пишет, а 2 в это время читает, см. race condition в википедии.

Процессы ОБЯЗАТЕЛЬНО должны иметь РАЗНЫЕ, НЕ ПЕРЕКРЫВАЮЩИЕСЯ области стека. Можно, конечно, выделить 1 стек под 2 процесса, но тогда эти процессы НИКОГДА не должны выполняться одновременно, иначе результат может быть ОЧЕНЬ НЕОЖИДАННЫМ.

- **Что нужно, чтобы запустить процесс?**

1. Объявить переменную типа proc_t, это дескриптор Вашего процесса. Лучше объявить ее как глобальную.
2. Выделить память под стек процесса: самый лучший вариант — объявить массив типа stack_t, размер массива — размер стека. Стек тоже лучше объявить глобально.

3. Написать «main» Вашего процесса. В принципе подойдет любая функция вида void some_main(void * arg). Все процессы, запускаются в специальной обертке, которая корректно завершает процесс в случае возврата из его «main».Инициировать дескриптор процесса вызовом proc_init, или proc_init_isr, пример:

Ахтунг!!!

Для инициализации процессов из main, следует использовать только proc_init_isr!!!

```
proc_init_isr(
		&my_proc,  /*  указатель на дескриптор процесса  */
		my_pmain, /* указатель на «pmain» процесса */
		my_sv_hook, /* указатель на «sv_hook» процесса */
		my_rs_hook, /* указатель на «rs_hook» процесса */
		my_main_arg, /* указатель на  аргумент */
		&my_stack[STACK_LENTH — 1],  /* указатель на дно стека, STACK_LENTH — длина стека */
		MY_PRIORITY, /* приоритет процесса, см. планировщик */
		MY_TIME_QUANT, /* квант времени, см. планировщик */
		IS_RT,  /* установить флаг PROC_FLG_RT, для процессов «жесткого реального времени», см. планировщик */
		);
```

4. Запустить процесс функцией proc_run, либо proc_run_isr, пример:

Ахтунг!!!

Для запуска процессов из main, следует использовать только proc_run_isr!!!

```
	proc_run(&my_proc);
```

Отличие между proc_run и proc_run_isr: последняя предназначена для запуска процессов из main и обработчиков прерываний. В результате вызова этих функций процесс начинает исполняться с того места, на котором его исполнение было остановлено.

- **Что писать в «pmain» моего процесса?**

Да все, что угодно! Я лишь расскажу об инструментах управления исполнением процесса, которые Вы можете использовать:

```
proc_reset_watchdog(); /* сбрасывает watchdog таймер вызывающего процесса «реального времени», если нужно остановить такой процесс — рекомендую вызвать эту функцию перед остановом */
proc_self_stop(); /* останавливает вызывающий процесс*/
sched_proc_yeld(); /* останавливает вызывающий процесс, передает управление следующему процессу с текущим уровнем приоритета, если вызвана из процесса реального времени, иначе перемещает вызвавший процесс в список отработавших свой квант времени и передает управление голове списка готовых к выполнению процессов*/
```

Кроме того, еще можно использовать критические секции, захватывать и освобождать семафоры и мьютексы, ждать сигналов, и сигналить другим процессам, см. средства синхронизации процессов.

И да, будьте осторожны со статическими переменными, они будут общими для всех запущенных экземпляров, доступ к ним необходимо организовывать только с помощью средств синхронизации процессов.

- **Как еще я могу управлять исполнением процессов?**

Еще есть 2 функции останова процессов:

```
proc_run_isr(&some_proc); /* Запускает любой процесс по указателю на его дескриптор, для применения в обработчиках преываний и критических секциях*/

proc_stop(&some_proc); /* Останавливает любой процесс по указателю на его дескриптор*/

proc_stop_isr(&some_proc); /* Останавливает любой процесс по указателю на его дескриптор, для вызова из критических секций и обработчиков прерываний.

НЕ РЕКОМЕНДУЮ использовать для останова вызывающего процесса*/

proc_restart(&some_proc);/* Перезапускает процесс сначала*/

proc_restart_isr(&some_proc);/* Перезапускает процесс сначала, для вызова из критических секций и обработчиков преываний. */

proc_set_prio(&some_proc, some_priority); /* NEW!!! Устанавливает приоритет процесса, вне зависимости от его текущего состояния, если процесс ждет захвата мьютекса запускает наследование приоритетов. */
```

2 *Планировщик*

Планировщик — это часть ОС, управляющая исполнением процессов, именно работающий планировщик дает возможность запускать несколько процессов на 1 процессоре.

Планировщик BuguRTOS производит переключение между процессами по времени или по вызову функции перепланирования.

- **Как это работает?**

Для работы планировщика выделяется аппаратный таймер (системный таймер), который генерирует прерывания с периодичностью, например, 1мС.

Обработчик прерывания системного таймера сохраняет все важные регистры процессора в стек текущего процесса, сохраняет значение указателя стека текущего процесса в его дескрипторе, вызывает функцию sched_schedule.

Функция sched_schedule выбирает новый процесс для исполнения и делает его текущим.

Обработчик прерывания системного таймера подменяет указатель стека на значение, сохраненное в дескрипторе нового текущего процесса, и восстанавливает из стека сохраненные значения регистров нового текущего процесса.

Дальше происходит возврат из обработчика прерывания системного таймера, и новый текущий процесс продолжает свою работу с того места, где она была прервана.

Функция sched_schedule устроена так, что время ее выполнения не зависит от количества выполняемых процессов (o(1)-планировщик).

Перепланирование происходит по вызову функции sched_reschedule().

Отличие перепланирования от обычной работы планировщика в том, что обработчик прерывания системного таймера передает управление следующему процессу в очереди, а при перепланировании происходит передача управления самому приоритетному процесу.

- **Как происходит выбор следующего процесса при планировании?**

В BuguRTOS все процессы делятся на процессы «реального времени» и процессы «общего назначения».

Первые предназначены для обработки событий, например, для организации «нижних половин» обработчиков прерываний. Большую часть времени процесс «реального времени» остановлен. При возникновении соответствующего события процесс запускается, обрабатывает это событие и останавливает себя.

Вторые предназначены для выполнения рутинных действий и большую часть времени работают.

Для разных видов процессов предусмотрены разные алгоритмы планирования:

- Для процессов «общего назначения» используется вытесняющая многозадачность с «честным» разделением процессорного времени между всеми процессами, при этом процесс, поставленный в очередь первым, начнет исполняться первым (fifo). Любой процесс может быть вытеснен другим более приоритетным процессом. Процессы с равными приоритетами исполняются по кругу (round robin).
- Для процессов «реального времени» используется многозадачность с полным вытеснением, процесс, поставленный в очередь первым начнет исполняться первым. Процесс «реального времени» вытесняет менее приоритетные процессы «реального времени». Если процессы «реального времени» имеют одинаковый приоритет, то они исполняются по кругу. Выполнение процесса «реального времени» продолжается до тех пор, пока он не будет вытеснен более приоритетным процессом, остановит себя, будет остановлен другим процессом, планировщиком или обработчиком прерывания.

- **Что такое приоритет процесса?**

Приоритет процесса определяет его очередность при выборе планировщиком следующего процесса для выполнения. В каждом дескрипторе процесса в BuguRTOS есть поле prio. Значение этого поля определяет

приоритет процесса, самому высокому проиритету соответствует значение 0, самому низкому — (BITS_IN_INDEX_T — 1), где BITS_IN_INDEX_T — разрядность индексного типа. Процесс idle (ему соответствует функция main проекта) работает с самым низким приоритетом.

- **Что такое квант времени?**

Квант времени это количество времени, в течении которого процесс может работать до того как будет остановлен планировщиком. Кванты времени выполняют 2 функции:

- наличие квантов времени дает возможность разделить процессорное время между процессами «общего назначения» по своему усмотрению;
- в случае ошибки в коде главной функции процесса и его «зависания» планировщик сможет остановить такой процесс и передать управление другому.

Планировщик по разному останавливает разные виды процессов: процессы «реального времени» удаляются из списка готовых к выполнению и их работа может быть возобновлена лишь после вызова proc_restart, либо proc_restart_isr; процессы «общего назначения» перемещаются в список процессов, отработавших свое время, их работа будет возобновлена самим планировщиком, когда все процессы общего назначения отработают свое время, и начнется следующий цикл выполнения процессов «общего назначения».

3 Средства синхронизации процессов

При разработке своего приложения Вы обязательно столкнетесь с проблемой обеспечения корректного доступа к общим ресурсам, а также с проблемой синхронизации процессов по времени и по событиям. Именно для этой цели в BuguRTOS предусмотрены средства синхронизации процессов.

Для контроля доступа к общим ресурсам предусмотрены критические секции, мьютексы и семафоры.

Для синхронизации процессов по времени и событиям — программные таймеры и сигналы.

Для передачи сообщений предусмотрен простой механизм IPC.

Ахтунг!!!

Процесс холостого хода никогда не следует останавливать! То есть из фцнкции **idle_main** нельзя **выходить**, в ней нельзя **захватывать мьютексы** и **семафоры**, в ней нельзя **ждать сигналы** и **IPC**

- **Критические секции.**

Кристическая секция — чать программы, при выполнении которой запрещены прерывания, таким образом при выполнении критической секции процесс не может быть остановлен. Критические секции могут быть вложены одна в другую.

На входе в критическую секцию вызывается функция enter_crit_sec(); на выходе — exit_crit_sec().

Код внутри критической секции должен работать БЫСТРО!!1 Ибо критические секции предназначены для кратковременного доступа к общим ресурсам.

- **Семафоры.**

В BuguRTOS реализованы счетные семафоры.

Суть работы семафоров такова: у каждого семафора есть счетчик количества процессов, которые могут одновременно его удерживать. Если этот счетчик больше нуля, значит процесс сможет захватить семафор и сможет использовать общий ресурс, который контролируется при помощи данного семафора; если счетчик равен нулю, значит процесс не сможет захватить семафор, будет остановлен и поставлен в очередь на захват семафора. Когда процесс освобождает семафор он дает возможность захватить этот семафор тем процессам, которые стоят в очереди на захват, естественно, если такие процессы есть, а если их нет, то происходит увеличение счетчика на 1.

Для того, чтобы использовать семафор нужно:

1. Объявить переменную типа sem_t;
2. Инициировать семафор, вызвав функцию sem_init или sem_init_isr:

Ахтунг!!!

Для инициализации семафоров из main, следует использовать только sem_init_isr!!!

```
sem_init( &my_sem, MAX_COUNT );
/* MAX_COUNT — максимальное количество процессов, которые могут удерживать семафор, если MAX_COUNT==1, то семафор работает как бинарный, если MAX_COUNT>1, то семафор работает как счетный*/
```

3. Для захвата семафора использовать sem_lock:

```
sem_lock( &my_sem );  /*Захват семафора my_sem*/
```

4. Для освбождения семафора использовать sem_unlock:

```
sem_unlock( &my_sem );  /*Освобождение семафора my_sem*/
sem_unlock_isr( &my_sem );  /*То же самое, но для использования в критических секциях и обработчиках прерываний.
Да, да, обработчик прерывания может освободить семафор, занятый процессом! */
```

5. Кроме того при захвате семафора устанавливается флаг состояния процесса PROC_FLG_SEM, который нужно обязательно сбросить после использования общего ресурса, вызвав функцию proc_flag_stop(PROC_FLG_SEM). Эта функция сбросит флаг и остановит вызвавший просцесс в том случае, если между захватом семафора и сбросом флага был запрос на останов процесса.

```
proc_flag_stop( PROC_FLG_SEM );  /* Сброс флага PROC_FLG_SEM и останов процесса при необходимости*/
```

Ахтунг!!!

Не рекомендую использовать семафоры для доступа к общим ресурсам!!! т.к. при этом нет гарантии освобождения такого семафора и соответственно общего ресурса!

Если Вы захватили семафор — освободите его обязательно, причем ДО того, как Ваш процесс «реального времени» будет остановлен.

- **Мьютексы.**

Для долговременного доступа к общим ресурсам предназначены мьютексы.

В каждом дескрипторе процесса есть счетчик захваченных ресурсов, если этот счетчик не равен нулю, то, в зависимости от конфигурации проекта, планировщик будет по разному управлять процессами жесткого реального времени: если не определен макрос CONFIG_HARD_RT, то планировщик не будет снимать с выполнения процессы, захватившие общий ресурс. Так обеспечивается гарантированное освобождение общих ресурсов.

При захвате мьютекса счетчик захваченных ресурсов увеличивается, и при освобождении уменьшается.

Ахтунг!!!

В BuguRTOS-0.7.x реализован протокол наследования приоритетов, в более ранних версиях был реализован протокол ICPP!!!

В остальном работа мьютекса аналогична работе бинарного семафора.

Для того, чтобы использовать мьютекс, потребуется следующее:

6. Объявить переменную типа mutex_t;
7. Инициировать мьютекс, вызвав функцию mutex_init или mutex_init_isr:

Ахтунг!!!

Для инициализации мьютексов из main, следует использовать только mutex_init_isr!!!

```
mutex_init( &my_mutex );
/* NEW!!! Теперь не надо объявлять приоритет мьютекса!*/
```

8. Для захвата мьютекса использовать mutex_lock:

```
mutex_lock( &my_mutex );  /*Захват мьютекса my_mutex*/
```

9. Для освбождения семафора использовать mutex_unlock:

```
mutex_unlock( &my_mutex );  /*Освобождение мьютекса my_mutex*/
```

- **Что выбрать?**

Если Вы встали перед выбором между критическими секциями, мьютексами и семафорами, вот несколько советов:

1. Если общий ресурс — переменная типа char/short/long и т.п., то следует применить критические секции.

2. Если общий ресурс — огромная структура и надо изменить в ней много полей, тогда следует выбрать мьютексы.

3. Если к общему ресурсу должны иметь доступ как процессы, так и обработчики прерываний, то однозначно следует применить критические секции, даже если это — огромная структура и надо изменить в ней много полей.

- **Сигналы.**

Сигналы нужны для того, чтобы оповещать ожидающие процессы об определенных событиях.

Если процесс ждет чего-то, то он может просто покрутиться в цикле, проверяя произошло ли это... Но зачем впустую нагружать процессор, когда есть сигналы?

Сигналы работают так: если процесс должен подождать какого-то события, то он останавливает себя и становится в очередь ожидания данного события, когда событие происходит, подается сигнал, который запускает один, или все ожидающие процессы.

Итак, для того, чтобы использовать сигнал, нужно:

1. Объявить переменную типа sig_t, это и будет сигнал.
2. Инициировать сигнал вызовом sig_init или sig_init_isr!:

Ахтунг!!!

Для инициализации сигналов из main, следует использовать только sig_init_isr!!!

```
sig_init( &my_sig );  /* Инициация сигнала*/
```

3. Если нужно подождать сигнал:

```
sig_wait( &my_sig );  /* Ставим вызывающий процесс в очередь ожидания сигнала*/
```

4. Если нужно запустить только голову списка ожидания:

```
sig_signal( &my_sig );  /* Запускаем голову очереди ожидания, можно вызывать только из кода процесса*/

sig_signal_isr( &my_sig );  /*то же самое, для применеия в обработчиках прерываний.*/
```

5. Если нужно запустить все процессы из списка ожидания:

```
sig_broadcast( &my_sig );  /* Запускаем все процессы из очереди ожидания, можно вызывать только из кода процесса*/

sig_broadcast_isr( &my_sig );  /*то же самое, для применеия в обработчиках прерываний.*/
```

Один и тот же сигнал может быть подан из нескольких источников, это можно использовать для того, чтобы ждать одно из множества событий.

- **IPC**

Для передачи данных между процессами или от обработчиков прерываний процессам в BuguRTOS есть простая реализация IPC. Механизм IPC в сочетании с семафорами и мьютесками предназначен для написания приложений с клиент-серверной архитектурой (а это один из способов избежать проблемы инверсии приоритетов).

Вот функции для работы с IPC:

```
ipc_data_t data = ipc_wait();  /* Останавливает процесс, переводит его в состояние ожидания получения данных по IPC, возвращает полученные данные*/

bool_t test = ipc_send( &waiting_proc, data );  /* Для передачи данных от процесса к процессу, проверяет дейчтвительно ли процесс waiting_proc ждет получения данных через IPC, возвращает true, если передача данных удалась*/

bool_t test = ipc_send_isr( &waiting_proc, data );  /* То же самое, но для передачи данных от обработчика прерываний к процессу*/

bool_t test = ipc_exchange( &waiting_proc, data_to_sent, &receive_container );  /* Функция обмена данными через IPC между процессами, пытается передать данные процессу waiting_proc, если передача удалась — переводит вызывающий процесс в состояние ожидания ответа от waiting_proc, возвращает true, если обмен удался, в receive_container записывается ответ от waiting_proc */
```

- **Программные таймеры.**

Программные таймеры нужны для синхронизации процессов по времени. Единица измерения времени в BuguRTOS — период генерации прерываний системного таймера. Вот список средств для работы со временем в BuguRTOS:

```
timer_t some_timer;  /* Объявляем переменную типа timer_t*/
CLEAR_TIMER( some_timer );  /*Макрос, сбрасывает программный таймер.
Можно использовать для инициации и собственно сброса таймера */
TIMER( some_timer );  /*Макрос, значение макроса — количество прерываний системного таймера с момента предыдущего сброса таймера.
Можно использовать во всяхих там if, while и пр.*/
void wait_time( timer_t time );  /*Функция ожидания в течение времени time, просто крутится в цикле указанное время.
Можно использовать для «длинных» программных задержек*/
```

4 *Обработчики прерываний*

Если обработчик прерывания не использует ресурсы BuguRTOS, то в этом случае для его описания следует воспользоваться встроенными средствами набора инструментов разработчика (например, в avr-libc это будут макросы ISR, SIGNAL, INTERRUPT).

В случае если обработчик прерывания использует ресурсы BuguRTOS, для объявления такого обработчика прерываний на архитектурах AVR и STM8 используйте макрос BUGURT_INTERRUPT:

```
BUGURT_INTERRUPT(INT0_vect){
    /*Тело обработчика прерывания*/
}
```

Этот макрос создает такую обертку вокруг Вашего обработчика прерываний, чтобы не нарушить целостность работу системы. В случае, если целевая платформа поддерживает несколько уровней приоритета прерываний, прерывания, объявленные с помощью макроса BUGURT_INTERRUPT, должны иметь наивысший приоритет.

Для Cortex-M макрос BUGURT_INTERRUPT не определен, так как его работу выполняют ядро процессора и контроллер прерываний, единственное ограничение: если в обработчике прерывания используются ресурсы BuguRTOS, то он должен иметь приоритет не выше, чем обработчики прерываний PENDSV и SYSTICK.

Начиная с версии 0.5.0 в BuguRTOS появилась поддержка вложенности прерываний. Для того, чтобы Ваша система могла работать с вложенными прерываниями нужно включить опцию CONFIG_PREEMPTIVE_KERNEL. При этом вложенность прерываний будет отключена по умолчанию, чтобы одно прерывание могло передать управление другому нужно в коде прерывания вызывать макрос KERNEL_PREEMPT() в каждом месте, где нужна передача управления другому прерыванию.

Так же следует организовать обработку прерываний в 2 этапа: «верхняя половина» — собственно сам обработчик прерывания, выполняет необходимый минимум работы по обработке прерывания, после чего запускает «нижнюю половину»; «нижняя половина» - процесс «реального времени», который делает все остальное и останавливает себя до следующего прерывания.

5 Заключение

Удачи, %username%, пишите хороший быстрый код, пользуйтесь BuguRTOS, ??????, PROFIT!!!